An experiment on the usefulness of design patterns: Detailed description and evaluation

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/694092
http://wwwipd.ira.uka.de/˜prechelt/

Technical Report 9/1997

June 16, 1997

Abstract

Advocates of software design patterns claim that using design patterns improves communication between software people. 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 74 participants of a university course on Java and design patterns. It finds that both hypotheses appear to be true, although the observed effects were relatively weak, presumably because the tasks were too simple in the experiment.

Contents

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

Chapter 1

Introduction

The fundamental principle of science, the definition almost, is this:
the sole test of the validity of any idea is experiment.
Richard P. Feynman

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 I will first discuss the general topic of the experiment (design patterns), then give a broad overview of the purpose and setup of the experiment, and finally describe related work.

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 possible 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. The appendices contain the handouts used in the experiment: questionnaire and program listings.

1.1 Design patterns

A few years ago, a group of people 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 and 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], advantages of design patterns are first that less experienced designers can produce better designs with patterns. Second, design patterns encourage recording and reusing best practices even for experienced designers, and third, design patterns can improve communication, both between designers and from designers to maintainers, by defining a common design terminology.

The first larger, 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 book enjoyed a giant success and resulted in heavily increased interest in design patterns.

Currently the number of design patterns reported in the literature is exploding and there are several conferences on the topic [7]. Some of the new design patterns are appealing, but most are only minor variations of known ones. 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 in software engineering, all this activity builds on belief — not knowledge — that the developments are useful. Currently, this belief grounds on intuitive judgement or at best on anecdotal evidence as reported by practitioners from the pattern community [1].

Our group believes that systematic tests of purported advantages have to be carried out in order to understand the mechanisms: 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 first controlled experiment for testing 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 avoids misunderstandings.

As mentioned above, such communication can occur at design time, implementation time, or during 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 designed using patterns. 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: Several subjects received the same program (Java 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 how the performance of group A was different from group B.

The experiment was performed with 74 student subjects in a single session of 2 to 4 hours. The tasks were based on two programs of 7 and 10 printed pages length; solutions had to be written on paper.

1.3 Related work

As far as we know, no scientific investigations of the assumptions underlying design patterns have yet been published. The only reports available are experience reports and anecdotal evidence from protagonists of the design pattern community [1].

1.4 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 any detail but instead relies on the original experiment materials (questionnaires, task sheets, program listings) that are printed in the appendix.

Chapter 2

Description of the experiment

Minds, like parachutes, only function when open.
Anonymous

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, software decay during maintenance is known as a crucial cost factor in software engineering.

The question that the experiment asks is the following: 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? Can maintenance be done quicker or 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 for describing 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 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.

	first with PD then w/o PD	first w/o PD then with PD
first Element, then Tuple	E^+T^-	E^-T^+
number of subjects	19	18
first Tuple, then Element	T^+E^-	T^-E^+
number of subjects	18	19

Table 2.1: The four experiment groups and their size. (E^+T^- stands for "first perform Element with PD, then perform Tuple without PD" etc.)

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 less compromise the design structure of the program. 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 it 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 77) 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 (31 lines of PD added to the 393 line program *Element*, 20 lines added to the 585 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 class of each error 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 number of points each subject received in the lab course, using stratified random sampling. The experiment was conducted semi-blindly, i.e., the subjects did not know in advance whether a program would contain PD or not, but there was no placebo.

2.3 Preparation: The JAKK course

In order to have enough subjects with sufficient ability and comparable background, we teached a course to "breed" our subjects. The topic of the course, called JAKK (Java/AWT-Kompaktkurs), was Java and AWT

(Abstract Window Toolkit), which attracted over 100 highly motivated students. The course lasted 5 weeks, with one 90-minute lecture and one exercise per week. The exercises were submitted on disk and were graded by the instructors.

We gave two lectures about Java itself, one about AWT, one about thread programming, and one for discussing frequent errors etc. The course exercises were chosen as follows: Exercise 1 was writing a clone of the Unix *head* program. This exercise was meant only for learning how to install and operate the compiler and learning basic Java syntax, file I/O etc.

Exercise 2 introduced the Java package concept and the Composite and Visitor design patterns. It was a program for computing whole/part-price-information for hierarchically structured computer systems and their components.

Exercise 3 was the introduction into AWT. The task was displaying a vector of numbers in two different formats (histogram and sign list) and keeping the displays up-to-date when the vector changed. This exercise introduced the Observer design pattern.

Exercise 4 was an introduction into programming with threads. The task was concurrently searching files for lines that were acronyms of the search string or words that were anagramms of the search string. The algorithms were to be implemented using the Template Method design pattern. The number of concurrent threads was to be limited in either of two user-selectable ways, using the Strategy design pattern to make the selection transparent. The program had a GUI, we required all concurrent activity to be properly synchronized and free of race conditions.

Exercise 5 introduced mouse handling and animation. It was a simple graphical reaction game and did not require the use of any particular design pattern.

Thus, our participants learned about and practiced 5 design patterns: Composite, Observer, Strategy, Template Method, and Visitor. Each of these was shortly introduced in the lecture and further motivated and explained in the exercise descriptions. The latter made quite precise prescriptions where and how to apply the design patterns, so actual practice with them was ensured. 60 percent of the course participants also visited a software engineering lecture course during the same semester where they learned about 30 design patterns in a theoretical fashion without hands-on experience.

Those course participants that appear as subjects in our experiments invested an average of 10.6 hours per week (median: 10 hours) over the 5 weeks of the course. A minimum of 60 percent of all points of the exercises was required for participating in the experiment. See Figure 2.1 for the distribution of course points of our experimental subjects. Exercises 2, 3, and 5 were worth six points each, exercise 1 had three points, and

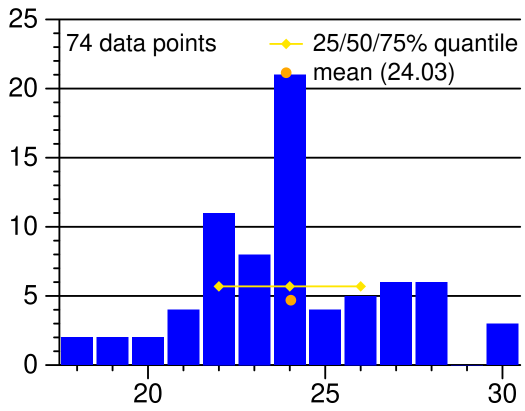


Figure 2.1: Distribution of points achieved during the Java course by our subjects. 30 points were possible, 18 were required for participating in the experiment.

exercise 4 had nine. Small errors in an exercise solution resulted in a partial reduction of points. Exercises 2, 4, and 5 had to be written from scratch, for exercise 1 and 3 similar or partial example programs were available to the students.

2.4 Experiment format and conduct

Participating in the experiment and showing reasonable performance was required to receive course credit for the Java course, but not all participants needed that credit.

We carried out the experiment in January 1997 on a Saturday morning, 10:00 to 14:00 hours. The experiment was conducted in a 350-seat lecture hall in a manner that was technically quite similar to an exam: Task descriptions and program listings were available on paper only and all results had to be delivered on paper as well.

The documents were handed out in five parts as described in Appendix A on page 40. Each subject could work in his personal pace and was given as much time as 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 40.

2.5 Experimental subjects

2.5.1 Education

74 subjects participated in our experiment. All of them were male Informatics master students and all but 10 of them held a Vordiplom (similar to B.Sc.) degree. On the average, these students were in their seventh semester at the university; see Figure 2.2.

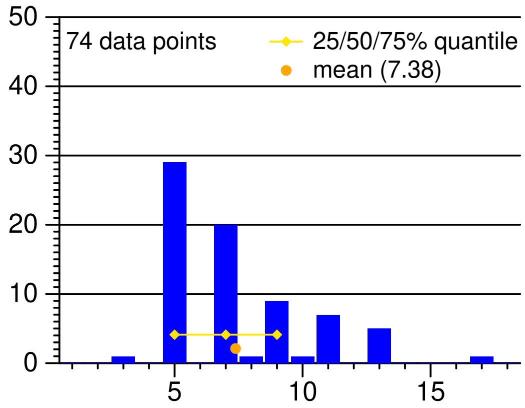


Figure 2.2: Distribution of semester numbers of experimental subjects.

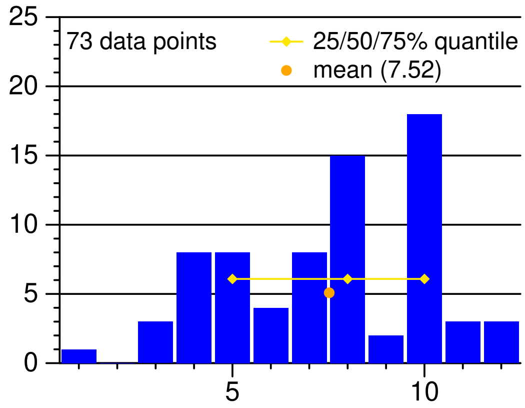


Figure 2.3: Distribution of years of programming experience of subjects.

2.5.2 Programming experience

These students had an average of 7.5 years of programming experience (Figure 2.3), 81% of them had written more or much more than 3000 Lines of Code (LOC) in their lifes (Figure 2.4 below). 69% had significant practical experience with object-oriented programming (Figure 2.5 below) and 58% had significant practical experience in programming graphical user interfaces (Figure 2.6 below). Our subjects had practice with an average of 4.6 different programming languages (Figure 2.7 below). The largest program ever written by our subjects had an average size of 3500 LOC (median: 2000 LOC) and 4.2 person months (median: 2 person months); see Figures 2.8 and 2.9 on page 12. 38% of the subjects had also previously participated in a

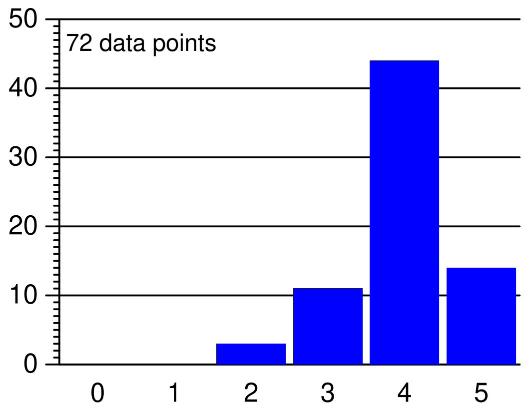


Figure 2.4: Distribution of previous programming knowledge and experience: 0=no knowledge, 1=only theoretical knowledge, 2=less than 300 LOC written, 3=less than 3000, 4=less than 30000, 5=more than 30000. The same encoding is used in the other two programming experience histograms.

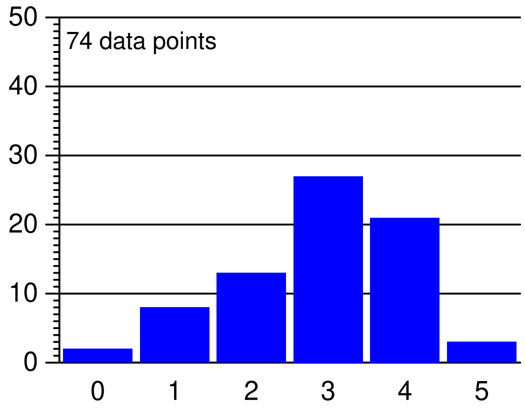


Figure 2.5: Distribution of previous experience in object-oriented programming.

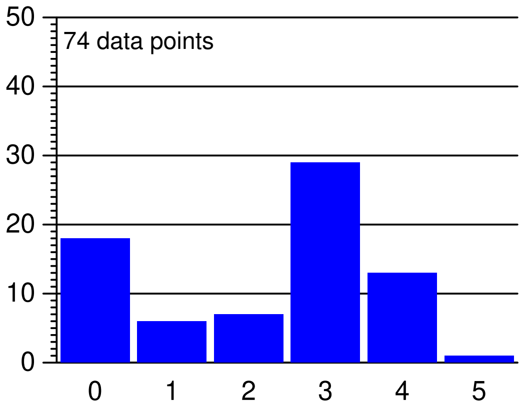


Figure 2.6: Distribution of previous experience programming graphical user interfaces (GUI).

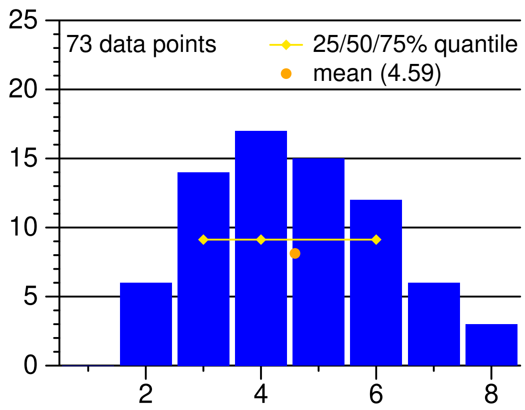


Figure 2.7: Distribution of number of programming languages previously used.

team software project and contributed an average of 2000 LOC (median: 1000 LOC) and 3.5 person months (median: 2 person months) to the total project size of on average 30000 LOC (median: 10000 LOC) and 51 person months (median: 10 person months); see Figures 2.10 to 2.13 below.

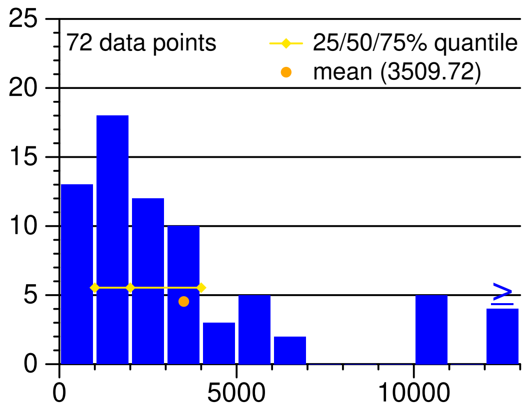


Figure 2.8: Distribution of size (in LOC) of largest program ever written alone.

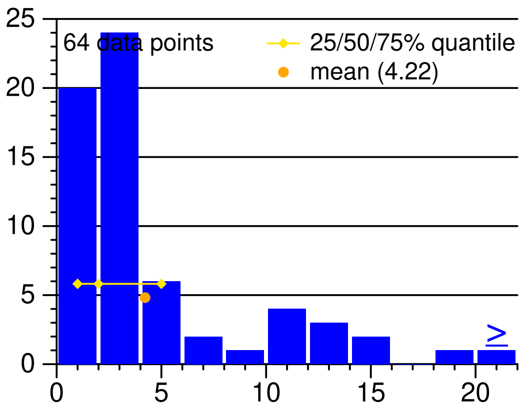


Figure 2.9: Distribution of size (in person months) of largest program ever written alone.

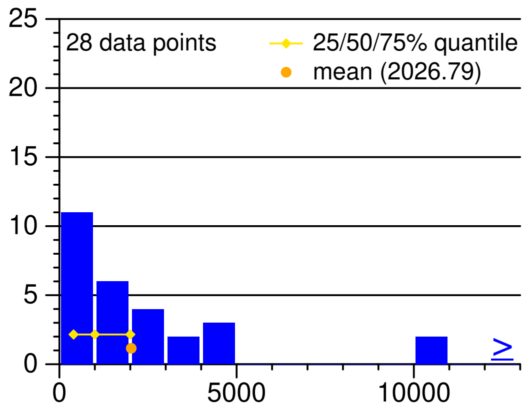


Figure 2.10: Distribution of size (in LOC) of subject's contribution to his largest team software project.

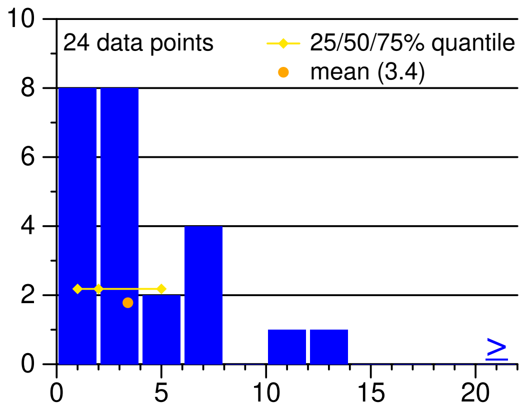


Figure 2.11: Distribution of size (in person months) of subject's contribution to his largest team software project.

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

On average, our subjects claimed to have reasonable or good knowledge of those patterns that were trained in the Java course (average grades of 1.82 to 2.16) and modest or little knowledge of other patterns (average grades of 2.94 to 4.00). Even for the former, there were always a few subjects, though, who admitted that their knowledge was insufficient; see Figure 2.14 below.

In the second questionnaire, we conducted an actual test of pattern knowledge; see Appendix A on page 40 for its exact form.

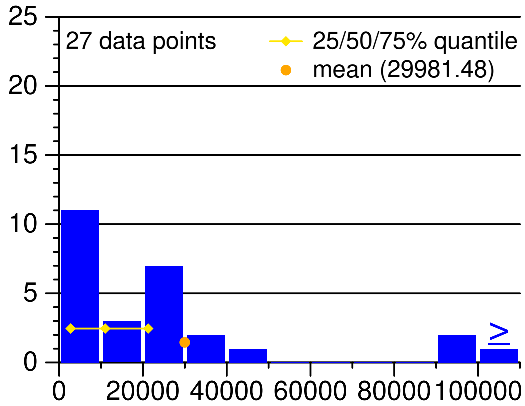


Figure 2.12: Distribution of size (in LOC) of largest team software project of subject.

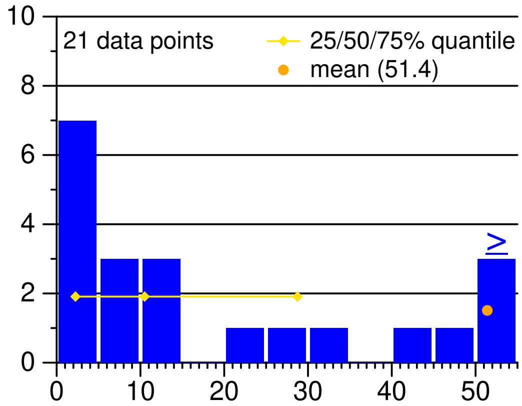


Figure 2.13: Distribution of size (in person months) of largest team software project of subject.

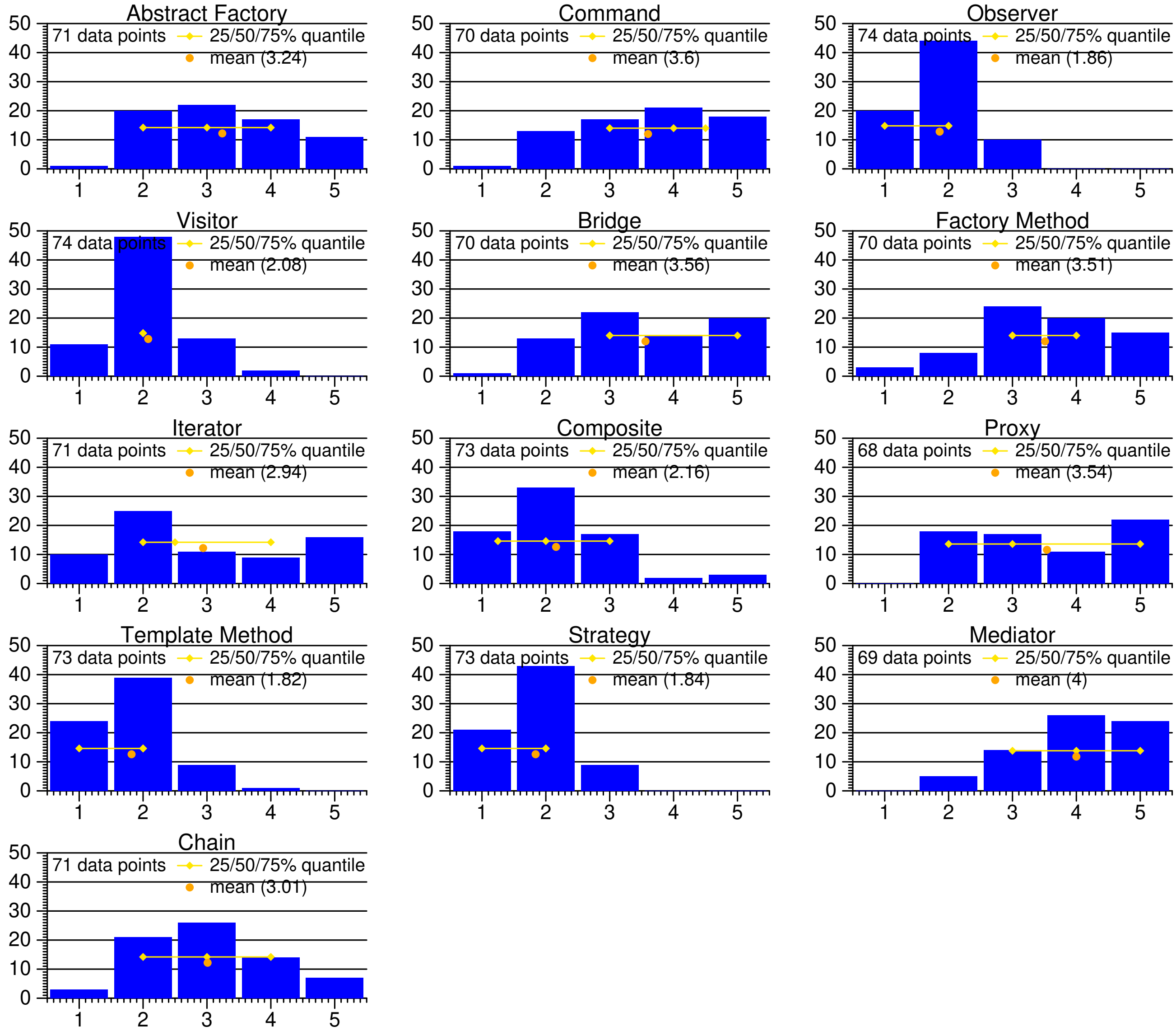


Figure 2.14: Subjective estimation of pattern knowledge. 1=understand very well, 2=understand well, 3=understand roughly, 4=begin to understand, 5=do not understand.

Operation	Command	Observer	Visitor	Composite	Template Meth.	Strategy	Mediator	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.

The first question of the test asked in which of 8 design patterns which of 6 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. The best possible result was 12 points, the worst possible was -53 points. The actual range obtained by our subjects was from -8 to 12 points. See Figure 2.15 for the point distribution. We find that more than half of the subjects obtained 7 or more of the 12 points, which is a quite

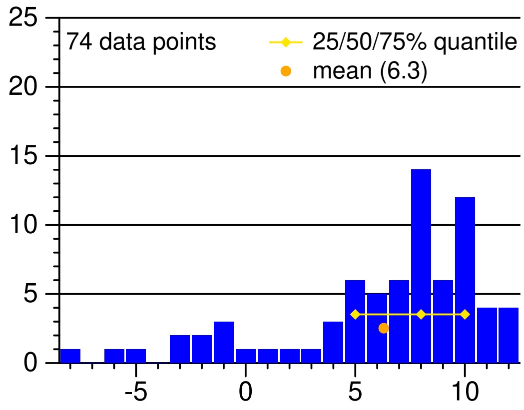


Figure 2.15: Distribution of points obtained in first question of pattern knowledge test.

reasonable level of pattern knowledge given the large number of possibilities for scoring negative points. One quarter of the subjects obtained 10 or more points.

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 to be visited; we counted 3 points in this case. Other answers such as using a Template Method (L) or an Iterator (G) have at least a little truth, but are not universal and therefore counted only 1 point. Often 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 or 2 points in these cases. Figure 2.16 below shows the point distribution for this question. We see that 60 percent of all subjects gave the right answer. 7 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 2.17 below. A few of the subjects made rather nonsensical suggestions such as using an Adapter (J) or Bridge (K) or Mediator (F) instead of thinking of a solution without a design pattern. As in question 1, however, overall pattern knowledge seems quite acceptable for most of our subjects.

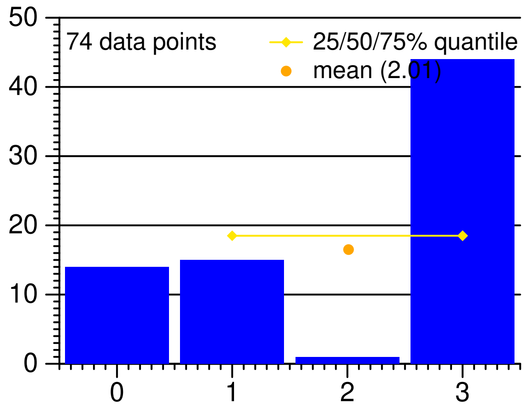


Figure 2.16: Distribution of points for second question of pattern knowledge test: What is the alternative to introducing a Visitor?

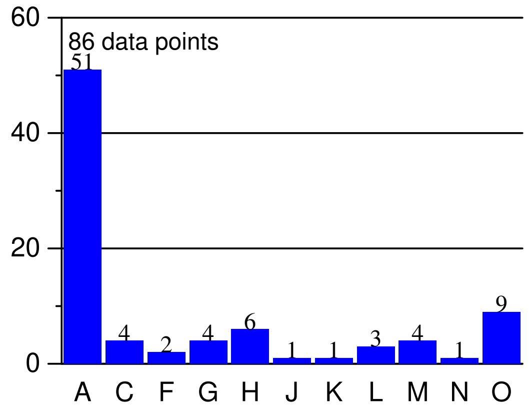


Figure 2.17: Frequency of different answers for this question: A:introduce new method, C:Strategy, F:Mediator, G:Iterator, H:Observer, J:Adapter, K:Bridge, L:Template Method, M:use subclasses, N:Chain of Responsibility, O:call available methods.

We did not evaluate at all the third of 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.

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. We assumed that four hours was a reasonable maximum, due to limits of the subjects' concentration ability. Therefore the tasks should not consume more than about one hour each for an average subject.
2. Only pen and paper were available as an infrastructure for both presentation of the tasks and solving.
3. 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.
4. The tasks could only employ those 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.

Constraint 2 was satisfied by presenting nicely pretty-printed program listings with line numbers and selecting change requests that required only small amounts of text to be written by the subjects.

For satisfying constraints 3 and 4, we selected one task containing an Observer pattern from a domain similar to exercise 3; it also contained a Template Method pattern, which had been practiced in course exercise 4. The other task was from a domain previously unknown to the subjects, but simple to explain and using a Composite and a Visitor pattern similar to course exercise 2.

2.6.3 Task “Tuple”

The “Tuple” program reads, stores, and displays structured tuples of firstname, lastname, and phone number using a graphical user interface (GUI). The store/display part is organized according to the Observer pattern. Two different displays are implemented, each in a different class. One class is completely coded, the other uses a superclass that contains a Template Method and only fills in the missing parts: 3 small methods for selecting, sorting, and formatting tuples.

The actual subtasks require the following: Finding two particular spots in the program (subtasks 1 and 2); introducing a new display class using the Template Method superclass (subtask 3); introducing another new display class from scratch (subtask 4). Please see the exact task descriptions of subtasks 3 and 4 in the appendix, as they are central for the experiment. As solutions, no method bodies were required from the subjects, but all other declarations had to be spelled out completely.

The two new display classes are analog to the existing ones, therefore the tasks are quite simple: Even if the current structure of the program is not understood, reasoning by analogy allows to solve the tasks by copying the structure of the corresponding class from the program. As we will see, this feature of the task makes it too simple and weakens the experiment results.

Whenever the button “more!” is clicked, the program will create a window “Read in Tuple” and read a single record (firstname, lastname, telephone number) typed into that window by the user. All these records are displayed together in other windows in two different styles. One window (“chronological”) shows the records in the order in which they were entered. The other window (“by lastname”) shows them sorted by lastname and in a somewhat different format.

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. As solutions, no method bodies were required from the subjects, but all other declarations had to be spelled out completely.

Subtask 3 is more difficult than subtasks 3 and 4 of “Tuple”. Still, however, 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 Methodological problems

Although the experimental design in itself is rather nice and clean, there is one important methodological problem in this experiment. This problem was induced by the constraint of working on paper only and concerns the subjects' resulting inability of compiling and testing their solutions. Under these circumstances many subjects deliver a "solution" that is very different from what they would deliver in a real software engineering context: The solution has flaws that would normally be corrected during compile or test.

In such cases, the quality of the solution cannot be judged in an appropriate way in the experiment and the time required for producing it is not realistic. This has to be kept in mind when interpreting the results of this experiment.

One may attempt to filter out the problematic subjects and consider the others only. Such a procedure is useful where applicable, but is limited by the ability to recognize the relevant cases.

2.8 Internal validity

There are two sources of threats to the interval validity of an experiment[1]: Insufficient control of relevant variables or inaccurate data gathering or processing.

As far as I 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, there was no mortality, environmental conditions were essentially the same for all subjects, and the counter-balanced experiment design controlled for any accidental group differences, learning, and sequencing effects.

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

2.9 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: 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.

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

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 more useful if 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 is obvious.
But don't look at it too carefully, or it will become unobvious
until you look at it for a very long time, then it becomes obvious again.
Anonymous Math Lecturer

For every problem there is a solution
which is simple, neat — and wrong.
Anonymous

This chapter presents and interprets the results of the experiment. The first section explains the means of statistical analysis and result presentation that I 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 postmortem 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"

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

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.

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 [8]. 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 in tables using absolute values and percentages.

For other data or for additional illustration I 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, we 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 8) 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 (with respect to a common wall clock) 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 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 3a, points 3b", for Tuple also "points 4a"), for the whole task ("all points"), and for the possibly PD-relevant subtasks ("relevant points", for Element this is 3a+3b, for Tuple it is 3a+3b+4a). Points are meant to characterize the quality of a solution, but this interpretation must be applied only with care, see Section 2.7 on page 17.

Productivity (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 31 subjects with high experience and 43 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 is dubious. It is possible that our test is no good measure of applicable pattern knowledge. There are 51 subjects with high pattern knowledge and 23 with low pattern knowledge.
3. Best solutions: Select only those subjects that produced a perfectly correct solution in the PD-relevant subtasks. For Element that means all subjects with 11 points in subtasks 3a+3b; for Tuple it means all subjects with 18 points in subtasks 3a+3b+4a. There are 22 subjects (15 with PD, 7 without) with best solutions for Element and 32 subjects (17 with PD, 15 without) with best solutions for Tuple.

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.

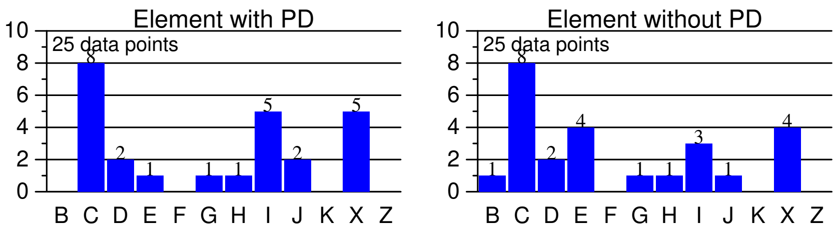


Figure 3.1: Frequency of different errors for subtask 1 of Element.
Codes: B:in StringElement.asString(), C:in AndElement.asString(), D:in OrElement.asString(), E:in StringElement.variants(), F:in AndElement.variants(), G:in OrElement.variants(), H:in main(), I:in StringElement.add(), J:in AndElement.add(), K:in OrElement.add(), X:other, Z:no answer.

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 3a, PD was supposed to be relevant. Therefore one would expect to find certain error classes more frequently in the group without PD (in particular classes B, Y, K, and M). Indeed, for B, Y, and M a difference is present as expected (see Figure 3.3 below) but none of them is statistically significant. In contrast to expectation, subjects without PD did *not* include the senseless auxiliary procedure more often than subjects with PD.

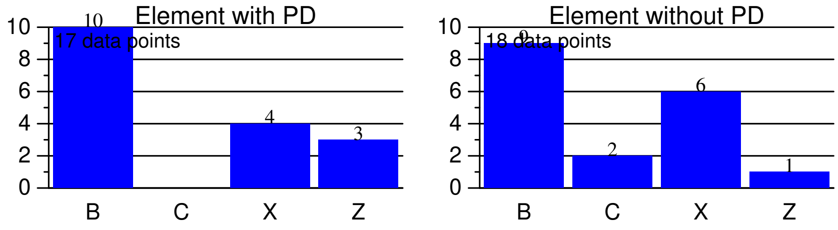


Figure 3.2: Frequency of different errors for subtask 2 of Element.
Codes: B:.length,.length(),.size or similar, C:no parens at variants(), X:other, Z:no answer.

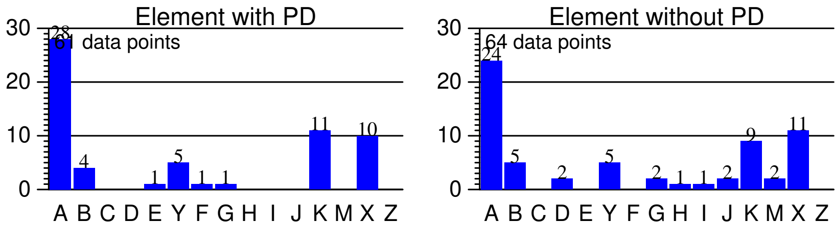


Figure 3.3: Frequency of different errors for subtask 3a (class construction) of Element.
Codes: A:best solution B:new method in xxElement classes, C:like A, but B also noted or present, D:like B, but A also noted, E:like B, but A also present, Y:constructed some different class, F:constructor or new method missing in Element, G:wrong constructor H:only verbal description such as “analog to Depth”, I:no integer instance variable, J:clearly superfluous instance variables, K:iterate() present, M:“extends Depth”, X:other, Z:no answer.
A, C, and E do not represent errors and costed no points. D, I, and J costed one point. B, F, K, and M costed two points. Y, G, H, and X were judged individually.

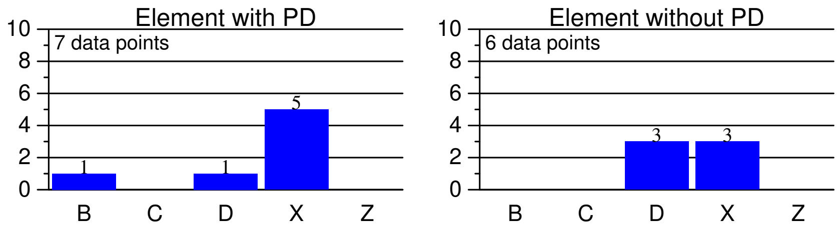


Figure 3.4: Frequency of different errors for subtask 3b (class use) of Element.
Codes: B:no Visitor object created (if solution with Visitor), C:counting method not called (if solution without Visitor), D:println missing, X:other, Z:no answer.
B, C, and D each costed two points.

	Task Element Variable	best	mean with PD P^+	 w/o PD P^-	means difference (90% confid.) I	signifi- cance p
1	points 1	2	0.97	0.97	$-38\% \ldots +38\%$	0.48
2	points 2	2	1.63	1.56	$-14\% \ldots +24\%$	0.35
3	points 3a	8	5.8	5.3	$-8.0\% \ldots +27\%$	0.18
4	points 3b	3	2.6	2.5	$-9.7\% \ldots +17\%$	0.33
6	all points	15	11.1	10.4	$-8.2\% \ldots +22\%$	0.23
7	relevant points	11	8.5	7.8	$-7.7\% \ldots +23\%$	0.20
8	— high pat.knwldg.	11	8.1	8.6	$-20\% \ldots +8\%$	0.25
9	— low pat.knwldg.	11	9.4	6.5	$+8.2\% \ldots +82\%$	**0.024**
10	— high experience	11	9.5	8.9	$-7.9\% \ldots +21\%$	0.23
11	— low experience	11	7.5	7.2	$-21\% \ldots +28\%$	0.40
12	time (minutes)	23	58.0	52.2	$-3.0\% \ldots +24\%$	**0.094**
13	— correct solutions	27	52.3	45.4	$-11\% \ldots +41\%$	0.17
14	— best 7	27	46.9	45.4	$-22\% \ldots +27\%$	0.41
15	— high pat.knwldg.	26	60.3	50.7	$+1.5\% \ldots +36\%$	**0.038**
16	— low pat.knwldg.	23	51.4	54.8	$-29\% \ldots +16\%$	0.37
17	— high experience	23	56.9	45.0	$+4.5\% \ldots +48\%$	**0.026**
18	— low experience	26	59.0	56.3	$-13\% \ldots +23\%$	0.33
19	points per hour	33	12.8	14.7	$-34\% \ldots +7.2\%$	0.14
20	— high pat.knwldg.	33	12.0	15.6	$-45\% \ldots -2.5\%$	**0.033**
21	— low pat.knwldg.	28	14.9	13.0	$-29\% \ldots +58\%$	0.27
22	— high experience	33	14.3	18.1	$-45\% \ldots +2.9\%$	**0.073**
23	— low experience	30	11.4	12.7	$-42\% \ldots +19\%$	0.27

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 points excluding subtasks 1 and 2. I and p were computed using resampling with 10000 trials.

In subtask 3b, PD might have been relevant, but as shown in Figure 3.4 above so few subjects made an error that no significant differences are visible.

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. We find the following results, summarized in Table 3.1:

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). More surprisingly, there is also no difference for the class use subtask (line 4) and only a rather small difference for the class construction subtask (line 3). As a result, the differences for all points (line 6) or relevant points (line 7, see also Figures 3.5 and 3.6 below) are also not significant.

The same is still true if we consider the relevant points of only those subjects with a lot of previous programming experience (line 10) or only those with little experience (line 11). A surprise occurs, though, if we use the amount of pattern knowledge as a filter (see section 3.2.1 on page 21): One should expect that PD was less useful for subjects with low pattern knowledge. In the experiment, however, subjects with high pattern knowledge received slightly less relevant points when PD was available (line 8), whereas subjects with *low* pattern knowledge received significantly more with PD than without (line 9)! Presumably subjects with low pattern knowledge wouldn't recognize the patterns without PD and make errors then, but can solve the tasks

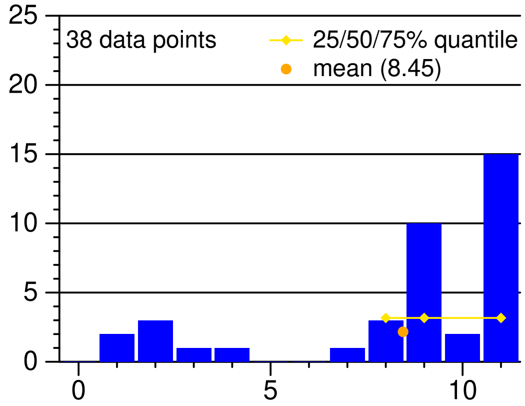


Figure 3.5: Distribution of “relevant points” obtained in task Element with PD.

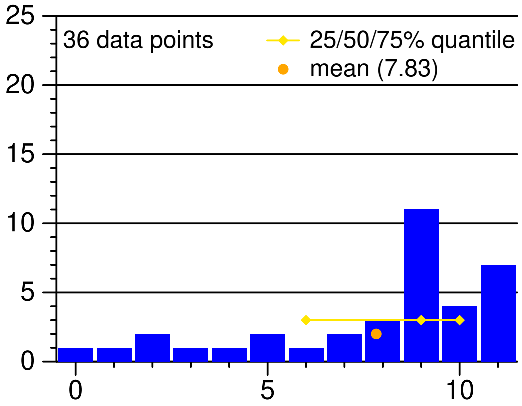


Figure 3.6: Ditto, without PD.

well once the patterns are recognized. In contrast, subjects with higher pattern knowledge may just have been more clever and imitated an existing class if no PD was available.

Summing up, PD had little influence on the number of points except for subjects that scored low in our pattern knowledge test, whom PD helped significantly.

Time: Another surprise: The time required for solving Element was significantly *higher* with PD than without (line 12, see also Figures 3.7 and 3.8). The difference becomes even more significant if we consider only the

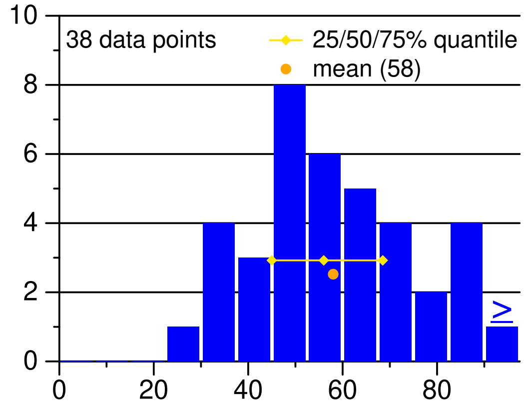


Figure 3.7: Distribution of time (in minutes) required for solving task Element with PD.

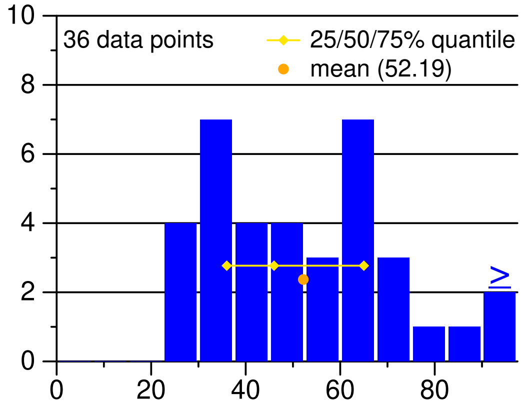


Figure 3.8: Ditto, without PD.

subjects with high experience or high pattern knowledge (lines 17 and 15), yet is not present for low experience or low pattern knowledge (lines 18 and 16). This effect may be a result of the methodological problem discussed in Section 2.7 on page 17. In this case it could be an artifact: the higher time investment for producing a good solution (which should be more frequent with PD and with high experience) is not compensated by the point reduction for a suboptimal solution. If this hypothesis is correct, we must find more correct solutions in the group with PD than in the group without PD. Indeed, there are 15 correct solutions in the PD group and only 7 in the non-PD group. This difference is significant ($\chi^2 = 3.55$, $p = 0.060$, Fisher exact $p = 0.051$). If we compare the times for only these subgroups C^+ and C^- with correct solutions, we find a much smaller difference (line 13), which further corroborates the hypothesis. The latter comparison is biased however, as the much larger group C^+ of correct solutions with PD probably contains also less talented subjects than the smaller group C^- without PD. Comparing only the presumably most talented 7 subjects of C^+ and C^- (according to lab course performance) we find essentially no difference ($p = 0.41$, line 14). I conclude that the difference of times is probably an artifact of the experiment.

This conclusion is also corroborated by the following observation. Of all measures obtained in the pretest that characterize the subjects, the one that best explains the variability in experiment performance is previous programming experience. If we plot programming experience against task completion time and against the resulting point score, we find that without PD task completion tends to be quicker the more experience a subject has (Figure 3.10), while the point score increases with experience (Figure 3.9), just as we would expect. With PD, on the other hand, the subjects with *medium* experience had the highest time investment, while the point score shows little variation and does *not* generally grow with increasing experience. The latter effect is probably caused by the class imitation approach being so successful.

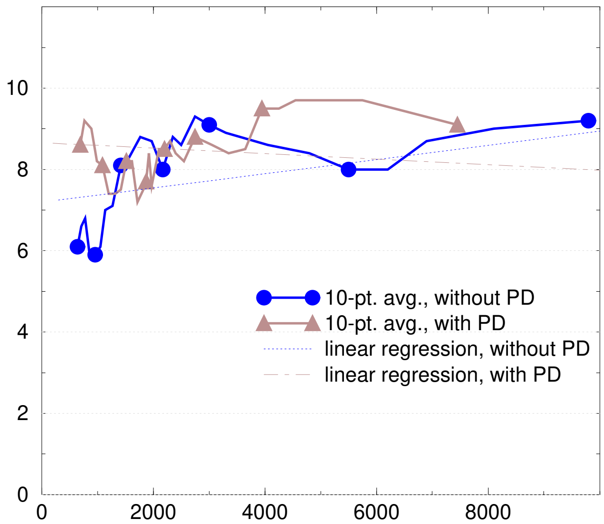


Figure 3.9: Experience versus points for Element. x-axis: size of largest program previously written by subject. y-axis: Number of points in Element subtask 3a, plotted as a running average of 10 data points.

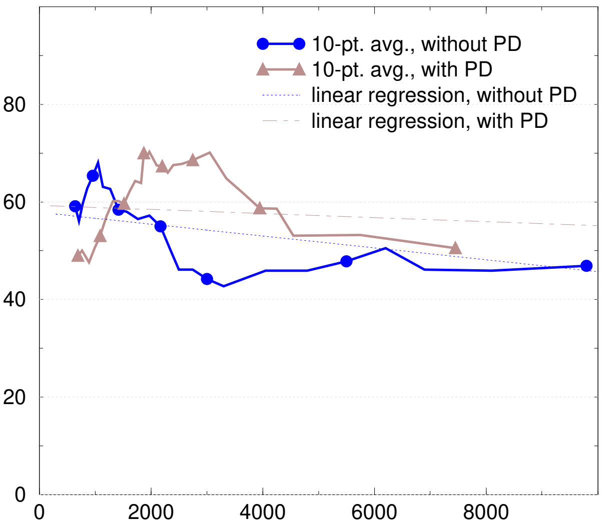


Figure 3.10: Experience versus time for Element. x-axis: size of largest program previously written by subject. y-axis: Time (in minutes) required for task Element, plotted as a running average of 10 data points.

For points per hour, the trend is analog to time alone.

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 the number of errors is also rather small as we can see in Figure 3.11. All differences are insignificant. For subtask 3a there are a few more errors in the group with PD as shown in Figure 3.12. The difference is insignificant, though, and the errors in the group without PD tend to be more serious ones.

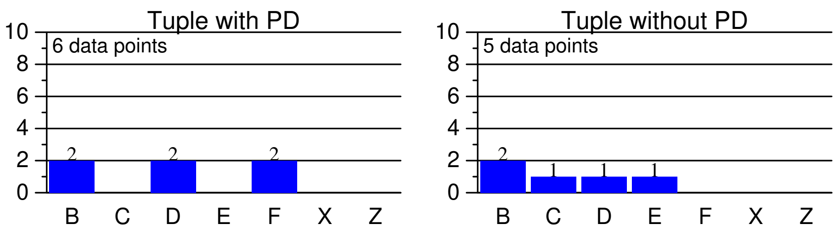


Figure 3.11: Frequency of different errors for subtask 2 of Tuple.
Codes: B:changed constructor of NTupleDisp2, C:in main(), D:multiple solutions, at least one wrong and one correct, E:completely wrong, F:adds a resize(), X:other, Z:no answer.

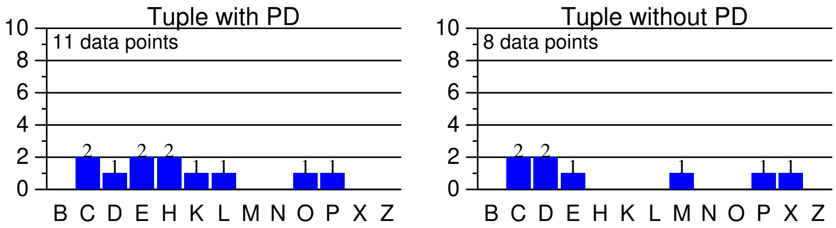


Figure 3.12: Frequency of different errors for subtask 3a (class construction) of Tuple.
Codes: B:no inheritance, C:removes “final” and inherits from NTTupleDisp2, D:wrong inheritance, E:constructor missing, H:format(), select(), or compare() missing, K:no Template Method used, L:wrong parameters, M:introduces new methods unnecessarily, N:answer too short and imprecise, O:uses multiple inheritance, P:multiple solutions at least one of which is correct, X:other, Z:no answer.
P does not represent an error and costed no point. C, H, K, L, and N costed one point. D, E, and M costed two points. B and O costed four points.

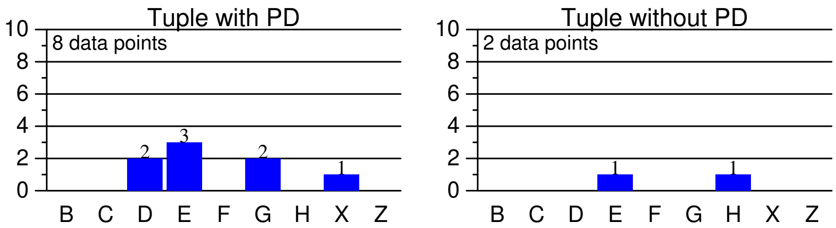


Figure 3.13: Frequency of different errors for subtask 3b (class use) of Tuple.
Codes: B:wrong constructor used, C:wrong argument for constructor, D:constructor called in wrong place, E:registration missing, F:registration in wrong place, G:task misunderstood, H:no line number given, X:other, Z:no answer.
H was not considered an error and costed no point. C costed one point. B, D, E, and F each costed two points. G costed four points.

Surprisingly, in subtask 3b (creation and registration of a new Observer), as shown in Figure 3.13, there are overall significantly more errors in the group with PD (8 versus 1, $\chi^2 = 6.64$, $p = 0.010$, Fisher exact $p = 0.011$). In particular subjects with PD forgot to register their new Observer more often (3 versus 1, $\chi^2 = 1.18$, $p = 0.28$, Fisher exact $p = 0.29$). The only explanation I can think of is that subjects without PD usually just imitated what was already present in the program for the existing observers, while subjects with PD created the code themselves and made errors doing so. If this explanation is correct, the difference occurs only because the tasks were too simple. Whatever, the difference is not significant. Subtask 4a again exhibits rather similar error profiles in both groups, with one exception (see Figure 3.14): Subjects without PD more often made inheritance errors (error classes D plus O, 9 versus 5, $\chi^2 = 1.16$, $p = 0.28$), but the difference is not significant.

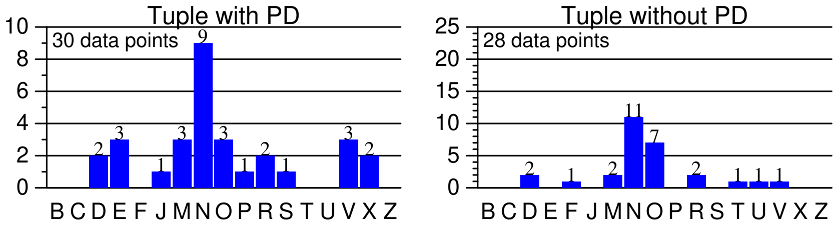


Figure 3.14: Frequency of different errors for subtask 4a (class construction) of Tuple.
Codes: B:no inheritance, C:removes “final” and inherits from NTTupleDisp1, D:wrong inheritance, E:constructor missing, F:wrong constructor, J:nonsense syntax, M:multiple solutions at least one of which is correct, N:inherits from TupelDispA, O:introduces new methods unnecessarily, P:introduces abstract class, R:TextArea missing, S:uses multiple inheritance, T:introduces abstract method, U:no implementation for all inherited abstract methods, V:introduces invert() method, X:other, Z:no answer. M and V do not represent an error and costed no point. C, J, R, T, and U each costed one point. D, E, N, O, and P each costed two points. S costed three points. B costed four points. F and X were judged individually.

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. We find the following results, summarized in Table 3.2 below.

Points: As expected, there is no difference in the number of points obtained in subtasks 1 and 2 (lines 1 and 2). There is also no difference for the class construction subtasks 3a and 4a (lines 3 and 5). There are similar numbers of correct solutions with and without PD for 3a (13 versus 15, $\chi^2 = 0.44$, $p = 0.51$) as well as 4a (5 versus 9, $\chi^2 = 1.16$, $p = 0.28$). In all of these cases, the subjects were rather close to the optimal number of points on average, which indicates that the tasks were easy. The above statements apply also to the total number of points and the relevant points (lines 6 and 7, see also Figures 3.15 and 3.16 below).

Surprisingly, subjects with PD received significantly *less* points in subtask 3b (line 4). Probably the subjects without PD relied more on imitating the code already present in the program which avoided mistakes more reliably.

Applying the filters (lines 8 to 11), we find that subjects with low pattern knowledge seem to be hampered somewhat by PD in the program (line 9). They probably tried to use PD for solving the program although they would have been better off if they had just imitated other classes as their colleagues without PD presumably did.

Time: For program Tuple, subjects with PD were significantly faster than subjects without PD (line 12, see also Figures 3.17 and 3.18 on page 30). The difference is largest for subjects with low pattern knowledge or low programming experience (lines 16 and 18). On the other hand the difference is insignificant if we consider only the correct solutions (line 13). Again, the reason is probably that without PD, most subjects relied on

	Task Tuple Variable	best	mean with PD P^+	mean w/o PD P^-	means difference (90% confid.) I	signifi-cance p
1	points 1	2	1.97	2.0	$-4.2\% \ldots +0.0\%$	0.36
2	points 2	3	2.8	2.8	$-5.1\% \ldots +9.2\%$	0.39
3	points 3a	8	7.6	7.6	$-5.7\% \ldots +6.5\%$	0.45
4	points 3b	4	3.6	4.0	$-16\% \ldots +0.8\%$	**0.031**
5	points 4a	6	4.8	4.8	$-12\% \ldots +12\%$	0.49
6	all points	23	20.8	21.1	$-6.0\% \ldots +3.3\%$	0.35
7	relevant points	18	16.1	16.3	$-8.0\% \ldots +4.0\%$	0.35
8	— high pat.knwldg.	18	16.6	16.3	$-3.3\% \ldots +6.7\%$	0.29
9	— low pat.knwldg.	18	15.2	16.4	$-23\% \ldots +5.0\%$	0.18
10	— high experience	18	16.8	16.6	$-4.9\% \ldots +6.8\%$	0.39
11	— low experience	18	15.7	16.1	$-12\% \ldots +6.6\%$	0.33
12	time (minutes)	19	51.5	57.9	$-22\% \ldots +0.3\%$	**0.055**
13	— correct solutions	26	55.7	52.8	$-12\% \ldots +22\%$	0.30
15	— high pat.knwldg.	20	53.0	57.1	$-22\% \ldots +7.8\%$	0.22
16	— low pat.knwldg.	19	48.9	60.2	$-35\% \ldots -2.0\%$	**0.032**
17	— high experience	20	45.2	52.8	$-34\% \ldots +5.2\%$	0.11
18	— low experience	19	55.1	62.5	$-25\% \ldots +0.9\%$	**0.064**
19	points per hour	66	27.6	24.7	$-6.1\% \ldots +29\%$	0.14
20	— high pat.knwldg.	63	27.3	25.8	$-14\% \ldots +26\%$	0.32
21	— low pat.knwldg.	66	28.0	21.7	$-1.3\% \ldots +65\%$	**0.058**
22	— high experience	63	32.7	27.7	$-8.3\% \ldots +43\%$	0.13
23	— low experience	66	24.6	22.0	$-10\% \ldots +35\%$	0.19

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 subtasks 1 and 2. I and p were computed using resampling with 10000 trials.

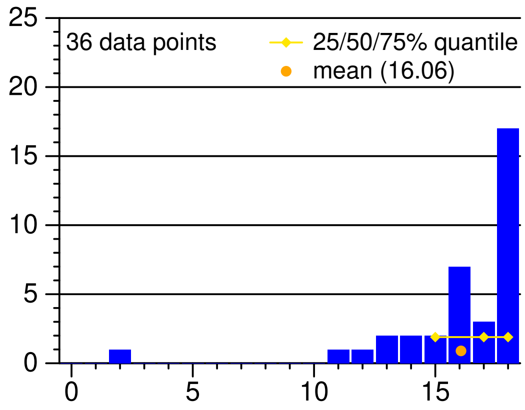


Figure 3.15: Distribution of "relevant points" obtained in task Tuple with PD.

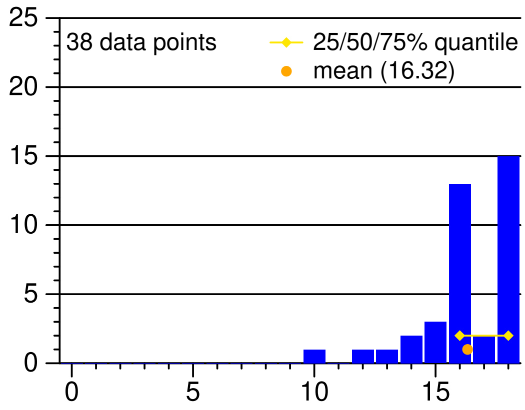


Figure 3.16: Ditto, without PD.

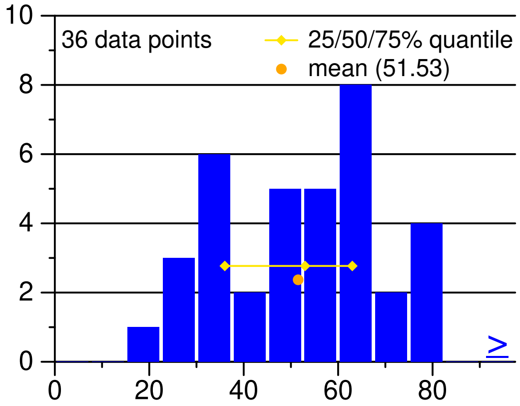


Figure 3.17: Distribution of time (in minutes) required for solving task Tuple with PD.

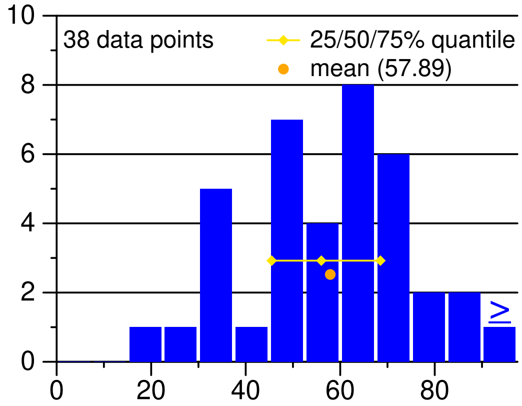


Figure 3.18: Ditto, without PD.

imitating the existing NTtupelDisp classes which worked both fast and safely. Combining points and times into productivity, we find small advantages for the PD group throughout (lines 19 to 23), but the difference is significant for subjects with low pattern knowledge only (line 21).

3.2.4 Learning effect

To assess a possible learning effect during the experiment, we evaluate the subgroups separately. For instance for the Element task, we compare 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 compare the differences of groups with and without PD for the first versus the second task. The results are shown in Table 3.3 below.

Points: With respect to "relevant points", there is no learning effect either for Element nor for Tuple: The difference between PD and no PD groups is significant neither in the first nor in the second task (lines 2, 3, 17, and 18) and the difference between first and second task is significant neither with PD nor without PD (lines 4, 5, 19, and 20). Only for Tuple with PD there is some advantage when it was performed as the first task instead of as the second (line 19).

Time: For time, the picture is entirely different. The time difference between with PD and without PD groups is similar in the first as in the second task for both Element and Tuple (lines 7 versus 8 and 22 versus 23), but there is a highly significant speedup from the first task to the second (lines 9, 10, 24, and 25). The speedup is present with PD as well as without and is particularly pronounced for Tuple (lines 24 and 25). As a result, there is also an improvement from first to second task in the points per hour measure for Tuple, both with and without PD (lines 29 and 30). There is no similar difference for Element (lines 14 and 15). In terms of point productivity, the differences between groups with and without PD became smaller from the first to the second task; for Tuple this meant a decreasing advantage of PD, for Element it meant a decreasing disadvantage. I must remind here that the points per hour measure is rather dubious, so the above result is hardly meaningful.

3.3 Underlying effects

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 3.20 and 3.19 on page 32. Not surprisingly, the subjects identified the patterns

	Variable	best	mean with PD or 1st	mean w/o PD or 2nd	means difference (90% confid.) I	significance p
	Task "Element":					
1	relevant points	11	$P^+ = 8.5$	$P^- = 7.8$	$-7.7\% \ldots +23\%$	0.20
2	— 1st task	11	$P^+ = 8.7$	$P^- = 7.9$	$-11\% \ldots +31\%$	0.22
3	— 2nd task	11	$P^+ = 8.2$	$P^- = 7.7$	$-17\% \ldots +28\%$	0.33
4	— with PD	11	$T_1 = 8.7$	$T_2 = 8.2$	$-14\% \ldots +28\%$	0.28
5	— without PD	11	$T_1 = 7.9$	$T_2 = 7.7$	$-19\% \ldots 24\%$	0.40
6	time (minutes)	23	$P^+ = 58.0$	$P^- = 52.2$	$-3.0\% \ldots +24\%$	**0.094**
7	— 1st task	26	$P^+ = 62.7$	$P^- = 56.1$	$-8.1\% \ldots +32\%$	0.16
8	— 2nd task	23	$P^+ = 53.3$	$P^- = 48.3$	$-7.6\% \ldots +29\%$	0.17
9	— with PD	28	$T_1 = 62.7$	$T_2 = 53.3$	$+0.5\% \ldots +34\%$	**0.046**
10	— without PD	23	$T_1 = 56.1$	$T_2 = 48.3$	$-5.9\% \ldots +39\%$	0.12
11	points per hour	33	$P^+ = 12.8$	$P^- = 14.7$	$-34\% \ldots +7.2\%$	0.14
12	— 1st task	24	$P^+ = 12.4$	$P^- = 14.4$	$-42\% \ldots +10\%$	0.16
13	— 2nd task	33	$P^+ = 13.2$	$P^- = 15.0$	$-43\% \ldots +19\%$	0.28
14	— with PD	23	$T_1 = 12.3$	$T_2 = 13.2$	$-36\% \ldots +20\%$	0.68
15	— without PD	30	$T_1 = 14.4$	$T_2 = 15.0$	$-36\% \ldots +29\%$	0.56
	Task "Tuple":					
16	relevant points	18	$P^+ = 16.1$	$P^- = 16.3$	$-8.0\% \ldots +4.0\%$	0.35
17	— 1st task	18	$P^+ = 16.6$	$P^- = 16.4$	$-4.9\% \ldots +7.6\%$	0.35
18	— 2nd task	18	$P^+ = 15.5$	$P^- = 16.3$	$-15\% \ldots +4.4\%$	0.23
19	— with PD	18	$T_1 = 16.6$	$T_2 = 15.5$	$-2.5\% \ldots +19\%$	0.12
20	— without PD	18	$T_1 = 16.4$	$T_2 = 16.3$	$-5.5\% \ldots +6.8\%$	0.42
21	time (minutes)	19	$P^+ = 51.5$	$P^- = 57.9$	$-22\% \ldots +0.3\%$	**0.055**
22	— 1st task	26	$P^+ = 57.3$	$P^- = 64.3$	$-16\% \ldots +1.4\%$	**0.096**
23	— 2nd task	19	$P^+ = 45.8$	$P^- = 51.5$	$-29\% \ldots +5.9\%$	0.14
24	— with PD	19	$T_1 = 57.3$	$T_2 = 45.8$	$+6.6\% \ldots +44\%$	**0.013**
25	— without PD	20	$T_1 = 64.3$	$T_2 = 51.5$	$+7.5\% \ldots +43\%$	**0.009**
26	points per hour	66	$P^+ = 27.6$	$P^- = 24.7$	$-6.1\% \ldots +29\%$	0.14
27	— 1st task	53	$P^+ = 24.8$	$P^- = 21.2$	$-3.1\% \ldots +41\%$	**0.082**
28	— 2nd task	66	$P^+ = 30.3$	$P^- = 28.3$	$-18\% \ldots +32\%$	0.33
29	— with PD	53	$T_1 = 24.8$	$T_2 = 30.3$	$-40\% \ldots +3.5\%$	**0.085**
30	— without PD	37	$T_1 = 21.2$	$T_2 = 28.3$	$-45\% \ldots -7.2\%$	**0.011**

Table 3.3: (left to right:) Name of variable, best result obtained by any subject, arithmetic average P^+ of sample of subjects provided with design pattern information or T_1 of sample for first task, ditto without pattern information (P^+ or T_2), 90% confidence interval I for difference $P^+ - P^-$ or difference $T_1 - T_2$ (measured in percent of P^- or T_2, respectively), significance p of the difference (one-sided). I and p were computed using resampling with 10000 trials.

more reliably when PD was given (Visitor: 33 versus 22, $\chi^2 = 11.05$, $p = 0.0009$, Fisher exact $p = 0.0008$. Composite: 34 versus 20, $\chi^2 = 16.39$, $p = 0.0001$, Fisher exact $p = 0.0000$). The number of spurious pattern identifications is similar with and without PD (13 versus 15, $\chi^2 = 0.09$, $p = 0.77$).

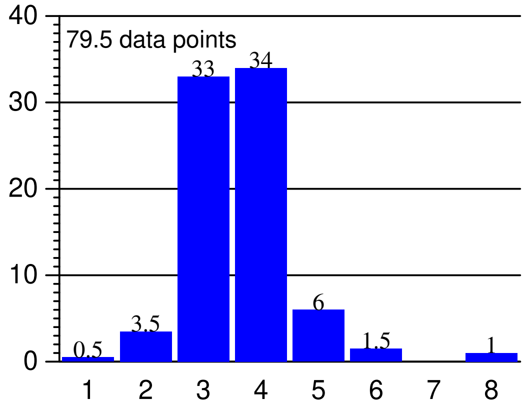


Figure 3.19: Number of times each pattern was checked as "found in Element" by subjects with PD. 1=Command, 2=Observer, 3=Visitor, 4=Composite, 5=Template Method, 6=Strategy, 7=Mediator, 8=Chain of Responsibility.

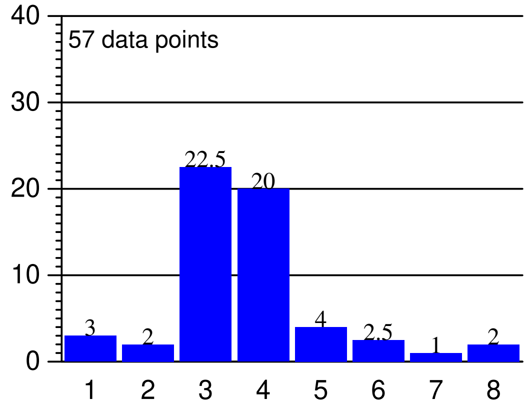


Figure 3.20: Ditto for subjects without PD. In all cases, checkmarks that were accompanied by question marks or question marks alone were counted as 0.5.

For Tuple, the patterns are also recognized more often with PD, but the difference is insignificant (Observer: $\chi^2 = 0.45$, $p = 0.50$, Template Method: $\chi^2 = 0.04$, $p = 0.84$, spurious patterns: $\chi^2 = 0.50$, $p = 0.48$).

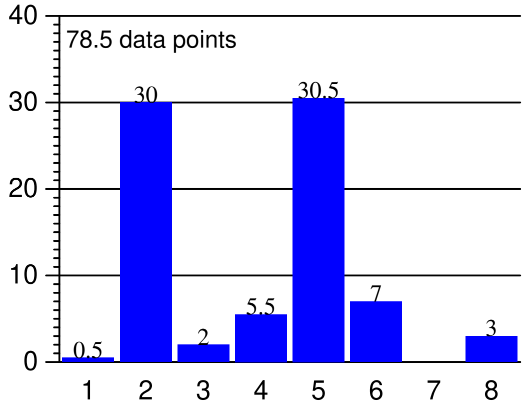


Figure 3.21: Number of times each pattern was checked as "found in Tuple" by subjects with PD. Encoding as in Figure 3.19.

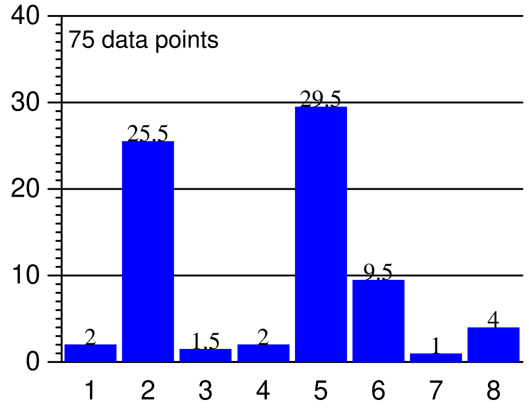


Figure 3.22: Ditto for subjects without PD.

As we see, subjects will not recognize patterns reliably without PD, even if they know only a small number of patterns and even if the program is not overly large or complex. On the other hand it seems that in the present case, recognizing the patterns was not necessary for all subjects as even with PD some of the patterns have been missed[2].

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

3.3.2 Problem solving method

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

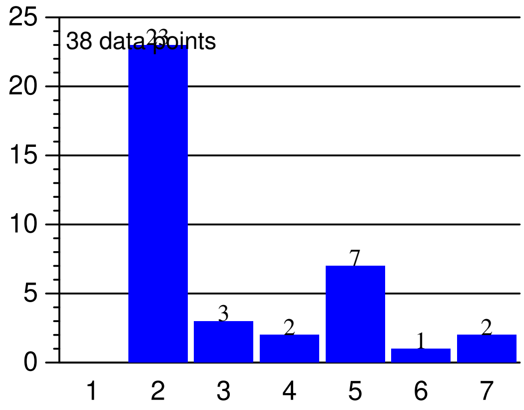


Figure 3.23: Whether, why, and when subjects actively searched for patterns in Element with PD. 1=No, unnecessary to know patterns; 2=No, patterns were documented; 3=No, found them immediately; 4=No, other reason; 5=Yes, right from the start; 6=Yes, when it became my only chance; 7=Yes, later for other reason.

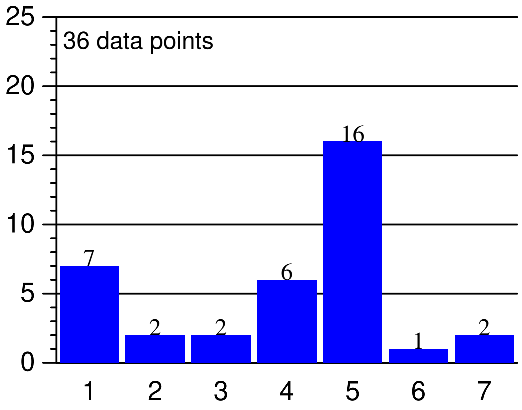


Figure 3.24: Ditto for subjects without PD.

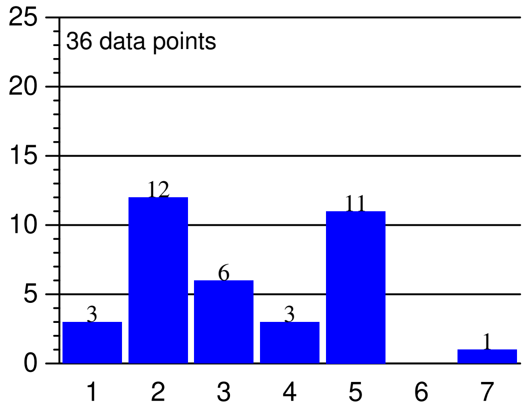


Figure 3.25: Whether, why, and when subjects actively searched for patterns in Tuple with PD. Encoding as in Figure 3.24.

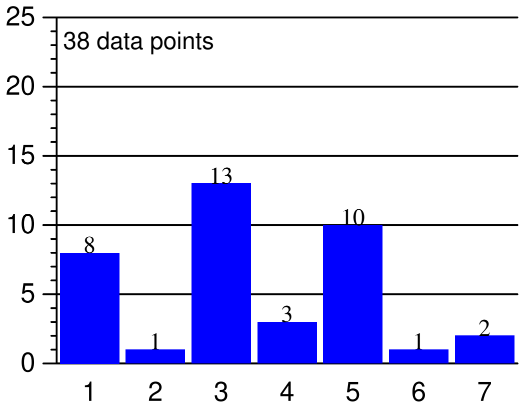


Figure 3.26: Ditto for subjects without PD.

The results are shown in Figures 3.24 to 3.25. For Element/Tuple, as many as 47%/66% of the subjects without PD did not search for patterns; another indication that the tasks were so simple that PD was not really required. Of these subjects, for Element a majority indicated that they did not feel the need to know about the patterns, whereas for Tuple a majority claimed to have recognized the patterns at once without searching. Even with PD, 26%/33% of the subjects said they searched for the patterns, most of them right from the start, which may indicate that they did not really expect to have PD in the program! Therefore, PD should be announced prominently. As for a learning effect, see Figures 3.27 and 3.28 below. Apparently only few subjects consciously learned that searching for the patterns might be helpful: a few more subjects claimed to have found the patterns

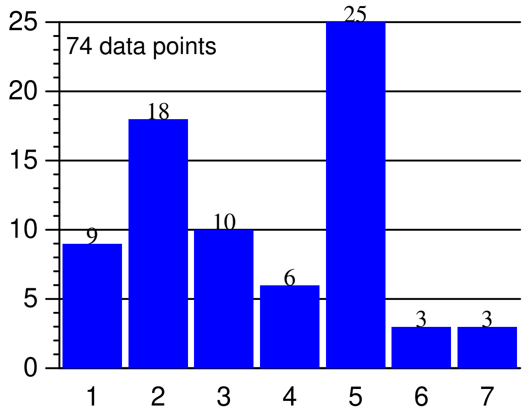


Figure 3.27: Whether, why, and when subjects actively searched for patterns in their first task. Encoding as in Figure 3.24.

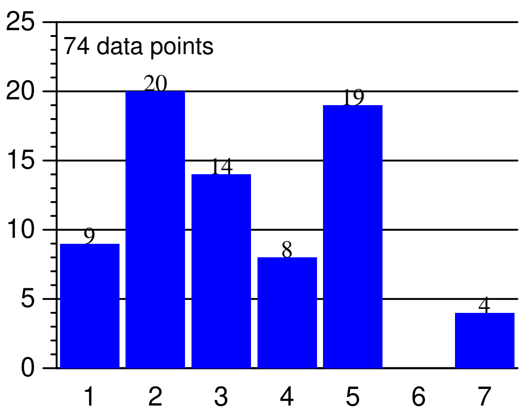


Figure 3.28: Ditto for second task.

immediately in the second task compared to the first (14 versus 10, $\chi^2 = 0.99$, $p = 0.32$) but a few less subjects started searching for patterns immediately (19 versus 25, $\chi^2 = 1.34$, $p = 0.25$).

3.4 Subjects' experiences

3.4.1 Difficulty of tasks

Our question how difficult the subjects found the tasks had rather surprising results: For the overall difficulty it did subjectively not make any difference whether PD was present or not; see Figure 3.29 below. This is true for Element as well as for Tuple. On average, both tasks were judged as simple, Tuple more so than Element. With regard to this question, there was no learning effect from the first task to the second.

How well the subjects were subjectively able to concentrate on the tasks is shown in Figure 3.30 below. Generally, our subjects could concentrate well; not much worse in the second task than in the first. Interestingly, the more difficult task (Element) resulted in a lower judgement than the easier one, but again 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 3.31 on page 36. Generally, the subjects had no high confidence in their solutions. Two thirds or more of them expected to have at least one error. Confidence decreased from the first task to the second, which is consistent with the perceived decrease in concentration. On the other hand, the presence or absence of PD made rather little difference.

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 3.32 on page 36. From all aspects, a majority of the subjects found previous pattern knowledge useful. 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.

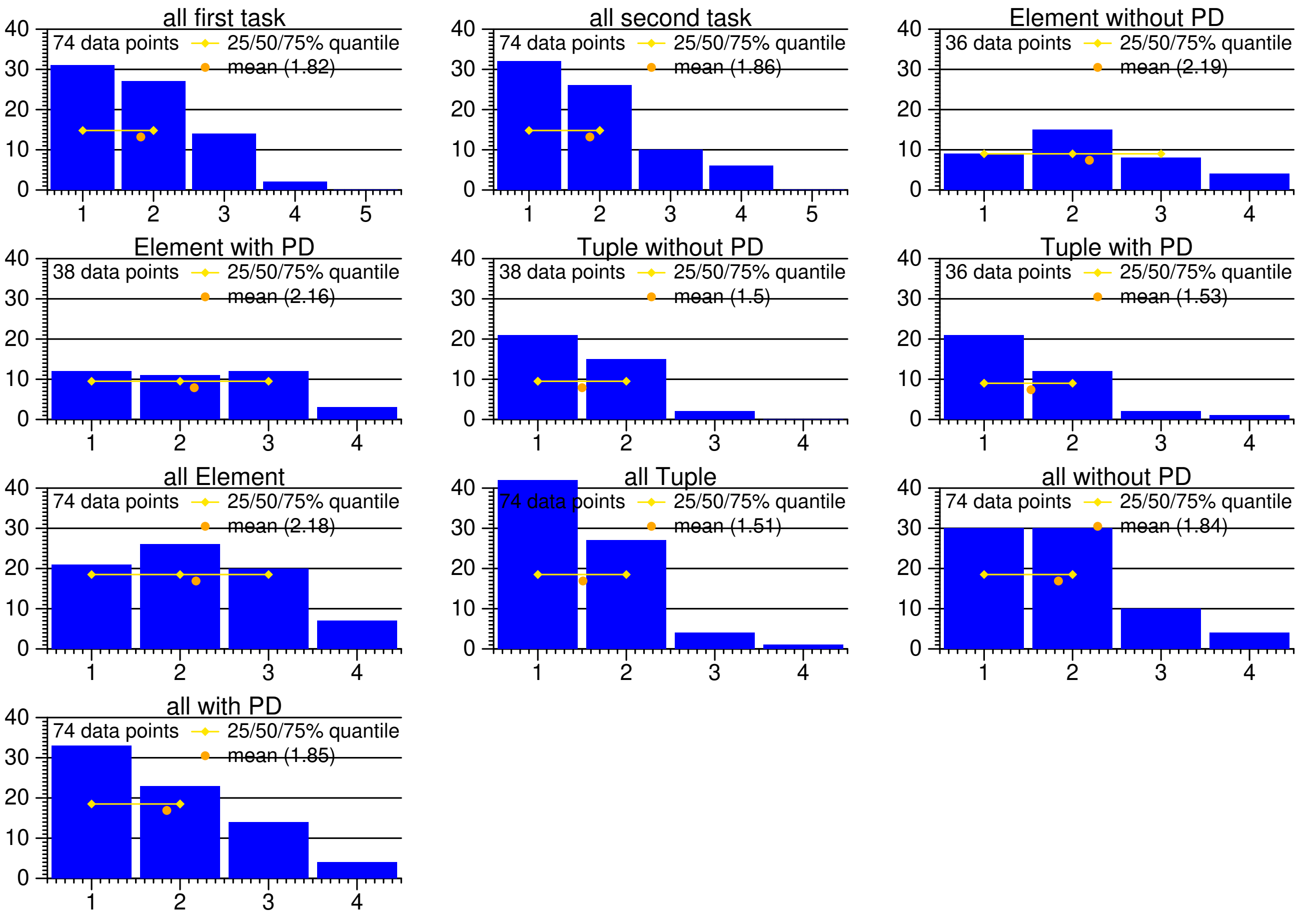


Figure 3.29: Subjective difficulty of tasks. 1=quite simple, 2=not too simple, 3=somewhat difficult, 4=difficult.

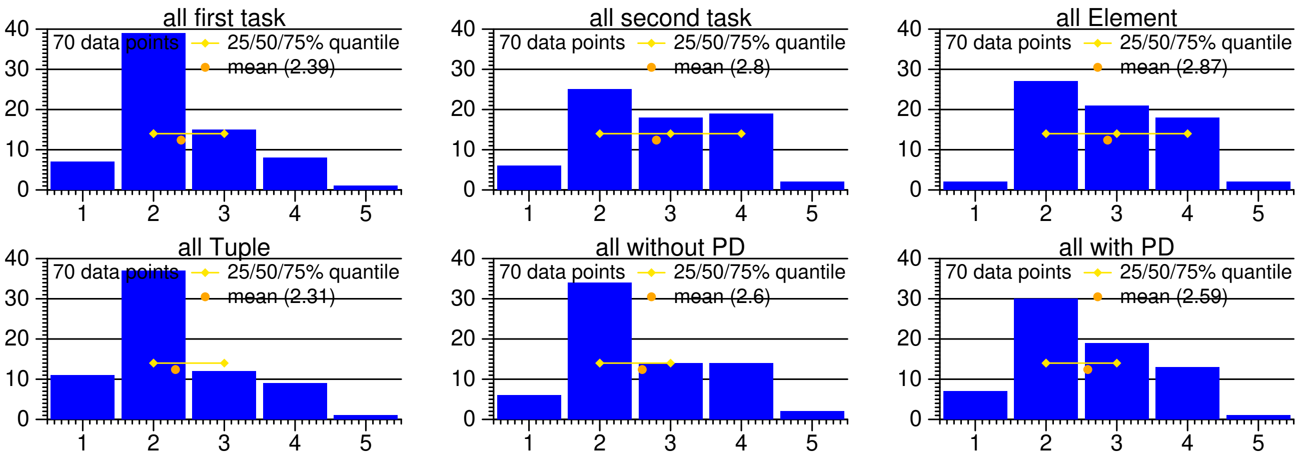


Figure 3.30: Subjective concentration ability during tasks. 1=very high, 2=high, 3=OK, 4=somewhat low, 5=low.

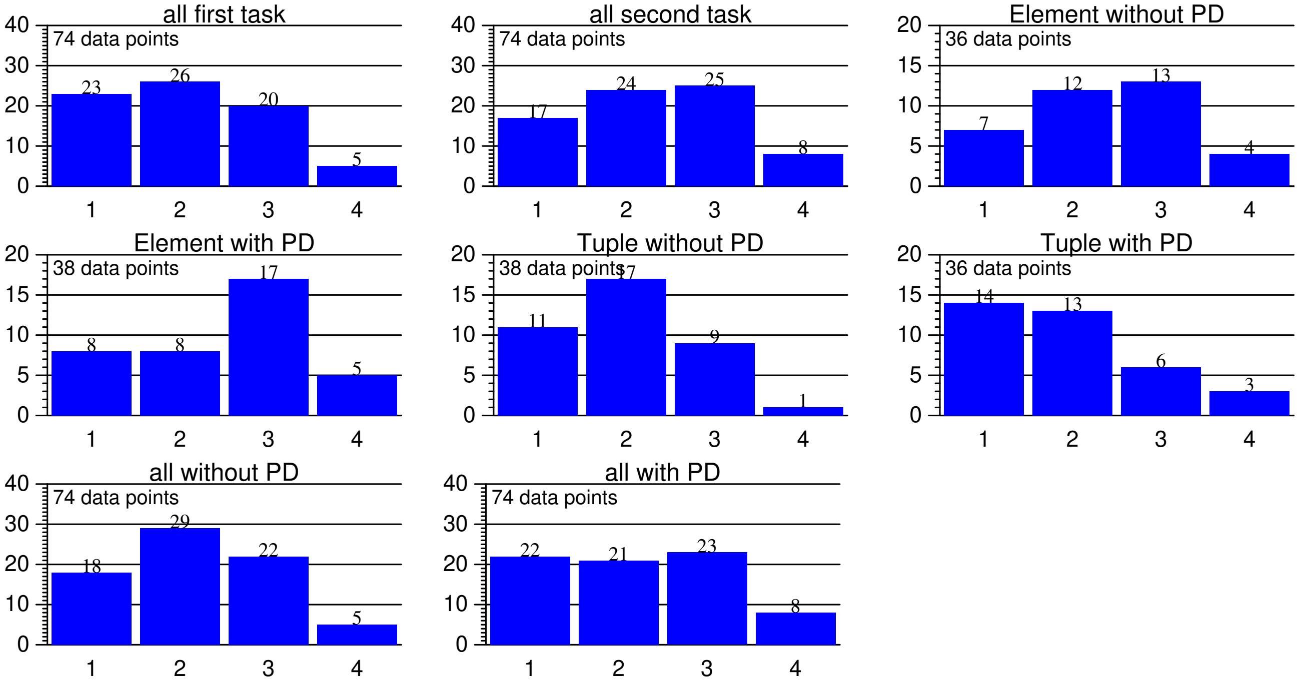


Figure 3.31: Subjects' estimate of errors in solutions. 1=none, 2=at most one,3=several, 4=don't know.

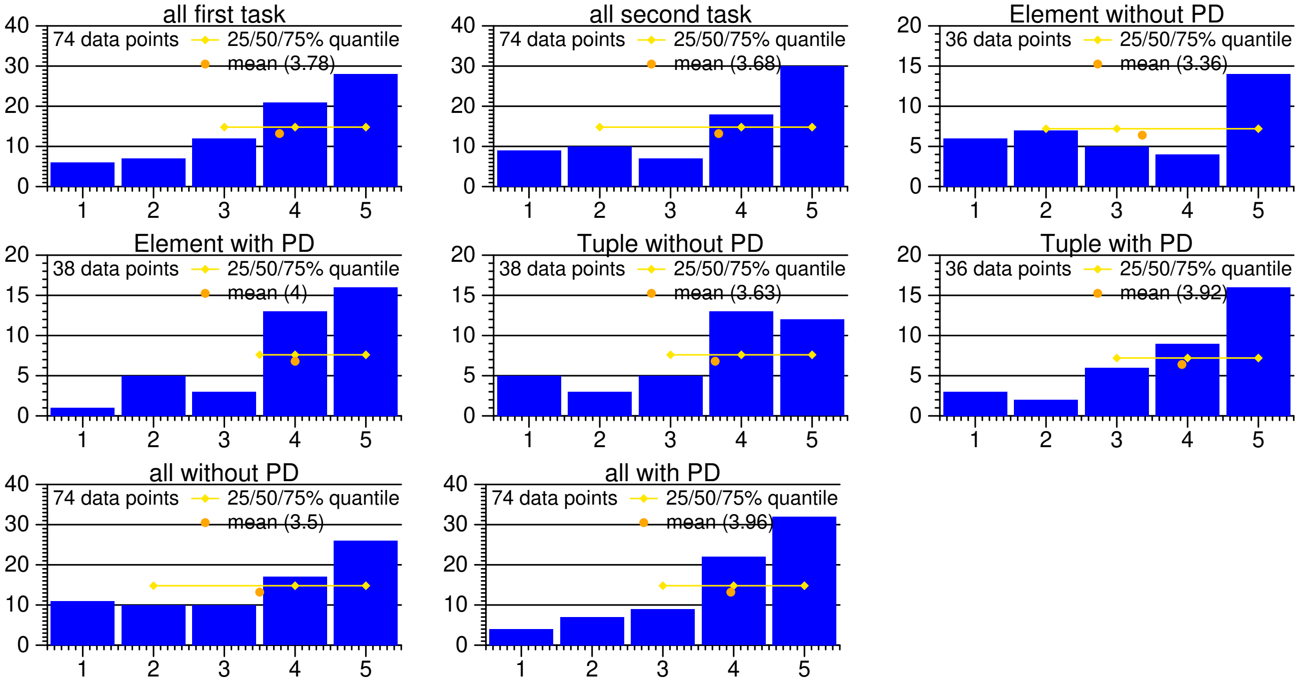


Figure 3.32: Subjective helpfulness of previous pattern knowledge for tasks. 1=no, 2=little, 3=am unsure, 4=yes, 5=yes much.

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 3.33. In principle, an answer to this question makes sense only for the cases with PD. Some of the

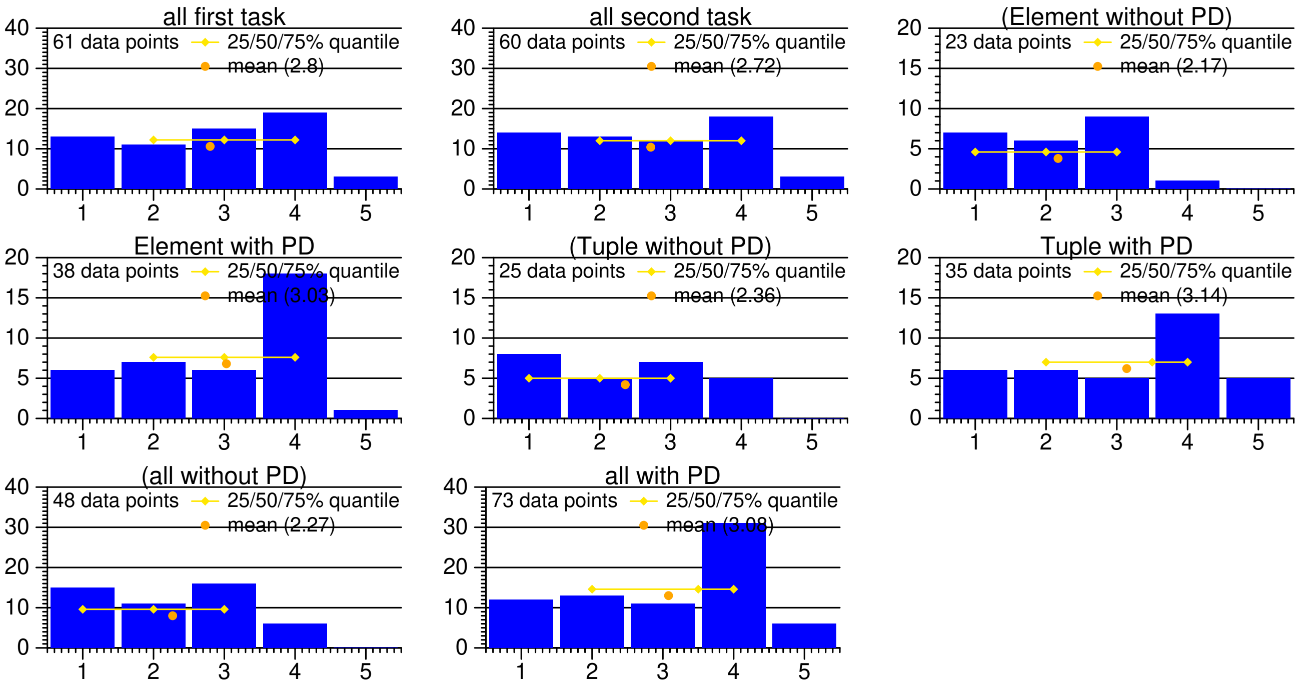


Figure 3.33: Subjective helpfulness of pattern documentation in the programs. 1=no, 2=little, 3=am unsure, 4=yes, 5=yes much.

subjects answered even without PD, though. Answers with question marks etc. were ignored. 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.

Chapter 4

Conclusion

Don't bite my finger,
look where I am pointing.
Warren McCulloch

The design of this experiment was *extremely* conservative; many design decisions biased the experiment towards *not* showing any effects from adding PD (see also the discussion of external validity in Section 2.9 on page 17):

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 will have made PD superfluous and will reduce its apparent benefits.
2. Furthermore, the subjects knew only few design patterns well; so even without PD they roughly knew what they could expect. In software production reality, a software engineer will gain a lot more information from PD because the catalog of possible patterns is larger.
3. The programs were rather small, so even without PD the subjects could achieve good program understanding within a reasonable time. Again, in reality PD will be more helpful for pattern-relevant tasks as the program understanding effort that it can save grows with the size of the program.
4. Due to the small program size, the pattern density in the programs was quite large. Therefore one could find the patterns quickly even if they were not documented. 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.
5. The programs were thoroughly commented, not only on the statement level, but also on the method, class, and program levels. Thus, the subjects had sufficient documentation available for program understanding even without PD. 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.
6. All pattern-relevant tasks in the experiment required adding functionality similar to existing functionality. So even without PD, the new parts could be derived by "imitating" analogous classes or statements. In the experiment, the imitation approach made PD completely superfluous. While imitation is sometimes possible in reality, most often it is not.

Given these circumstances, in particular the last one, 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 support both of our hypotheses (see Section 2.1 on page 7):

For the Tuple task, there were no significant differences in the number of correct solutions, but subjects with PD were significantly faster on average, thus supporting hypothesis H1 that PD can speed up task completion.

For the Element task, the number of correct solutions was significantly higher in the group with PD, thus supporting the hypothesis H2 that PD reduces the number of mistakes during maintenance. PD allowed less talented subjects to produce a correct solution, which is a rather impressive result indeed.

We therefore recommend that usage of design patterns routinely be documented in program source code.

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

Thanks to Oliver Gramberg, Hendrik Lock, Barbara Unger, and in particular Michael Philippsen for their contribution in teaching the course, to Michael Philippsen and Barbara Unger for helping design and conduct the experiment, to Barbara Unger for helping with the evaluation, to Michael Philippsen for reviewing a late draft of this report, and to Walter Tichy for suggesting the whole enterprise of a pattern validation experiment.

Appendix A

Questionnaire

There is nothing new under the sun,
but there are lots of old things we don't know yet.
Ambrose Bierce

This appendix contains the original questionnaire administered to the subjects. The questionnaire was handed out in 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 listing)
4. the explanation of the second task (handed out together with the corresponding program listing)
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.

There are two versions of the questionnaire in this report: The original German version as given to the subjects and an English translation (starting on page 60). The latter was kept as close to the original as possible and therefore is a little rude at some points.

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 14
- Pattern question b:
  introducing a new method in all classes to be visited
- Tuple subtask 1:
  353 (in `NTTupelAnz1.neuesTupel()`, corresponds to line 345 in `NTTupleDisp1.newTuple()` of the English version)

- Tuple subtask 2:
  390 (in `TupelAnzA.TupelAnzA()`, corresponds to line 382 in `TupleDispA.TupleDispA()` of the English version)

- Tuple subtask 3a:

```
final class NTTupleDispA extends TupleDispA {
  NTTupleDisp3(String name) {}
  String format(Tupel a) {}
  boolean select(Tupel a) {}
  boolean compare(Tupel a, Tupel b) {}
}
```

  (All method bodies would be exactly like in `NTTupleDisp2` except for `select()`.)

- Tuple subtask 3b:

```
  TupleDisplay disp3 = new NTTupleDisp3("nonlocal");
  mainwindow.allTuple.newDisplay(disp3);
```

- Tuple subtask 4:

```
final class NTTupleDispR extends TupleDisplay {
  private TextArea textarea;
  NTTupleDispR(String name) {}
  synchronized void newTuple(Tuple t) {}
  synchronized void newTuples(Vector newtuples) {}
}
```

  (All method bodies would be exactly like in `NTTupleDisp1` except for `newTuple()`.)

- in Element subtask 1:
  211 (in `AndElement.variants()`, both German and English version)

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

- in Element subtask 3b:

```
class CountVariants implements ElementAction {
 int variants = 0;  // number of variants in visited Element
 CountVariants(Element e) { e.perform(this); }
 public void stringAction(StringElement e) { variants = 1; }
 public void andAction(AndElement u) {
   int allVariants = 1;
   Enumeration n = u.elems.elements();
   if (!n.hasMoreElements()) {  // empty AndElement has 0 variants,
      variants = 0;             // not 1!
      return;
   }
   while (n.hasMoreElements()) {
```

```
        Element e = (Element)n.nextElement();
        e.perform(this);    // compute variants for one subelement
        if (variants > 0)   // ignore empty AndElements!
           allVariants *= variants;  // otherwise multiply variants
     }
     variants = allVariants;
   }
```

The above solution also contains the method bodies, which were not required in the experiment task. However, one has to think about what the bodies would look like or would be tempted to include an `iterate()` auxiliary method as in the pre-existing `Depth` visitor, although such a method is not useful here.

A.1 Original questionnaire

Anleitung und Fragebogen für das Java/AWT-Experiment

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

16. Juni 1997

Matrikelnummer:

(Hier bitte nichts eintragen)

Aufgabe	Punkte
E	
S	
Ms	
Ma	
Mb	
Mc	
Md	
T1	
T2	
T3a	
T3b	
T3c	
T4a	
T4b	
E1	
E2	
E3a	
E3b	
E3c	

Fragebogen Teil 1: Zu Ihrer Person

Bitte füllen Sie diesen Teil des Fragebogens vor dem weiteren Lesen und vor dem Beginn der Versuchsdurchführung durch Eintragen und Ankreuzen aus. *Alle Angaben werden vertraulich behandelt.*

Vollständige und korrekte Angaben (bitte in Druckbuchstaben!) sind wichtig für die Genauigkeit der wissenschaftlichen Aussage, die aus dem Experiment gewonnen werden kann. Deshalb sind wir sehr dankbar, wenn die Angaben möglichst keine Lücken haben.

____________________ ____________________
Name Vorname

_______________ __________
Matrikelnummer Geschlecht: m/w

_______________ __________
heutiges Datum Uhrzeit

☐ Ich bin ________________student/in im _____-ten Fachsemester,
Fach
☐ mit Vordiplom, ☐ noch ohne Vordiplom.

☐ Ich habe die *in diesem Semester* die Vorlesung „Softwaretechnik“ gehört und dort diverse Entwurfsmuster kennengelernt.

Ich hatte vor dem Java-Kurs insgesamt folgende Programmiererfahrung:

☐ nur theoretische Kenntnisse.

☐ weniger als 300 Programmzeilen selbst geschrieben.

☐ weniger als 3.000 Programmzeilen selbst geschrieben.

☐ weniger als 30.000 Programmzeilen selbst geschrieben.

☐ mehr als 30.000 Programmzeilen selbst geschrieben.

Ich programmiere seit ungefähr _____ Jahren und habe dabei überwiegend folgende Sprachen benutzt (in absteigender Reihenfolge der Häufigkeit):

____
Sprache 1, Sprache 2, …

Ich hatte vor dem Java-Kurs folgende Erfahrung mit objekt-orientierter Programmierung:

☐ keine Kenntnisse.
☐ nur theoretische Kenntnisse.
☐ weniger als 300 Programmzeilen selbst geschrieben.
☐ weniger als 3.000 Programmzeilen selbst geschrieben.
☐ weniger als 30.000 Programmzeilen selbst geschrieben.
☐ mehr als 30.000 Programmzeilen selbst geschrieben.

Ich hatte vor dem Java-Kurs folgende Erfahrung mit der Programmierung graphischer Benutzerschnittstellen:

☐ keine Kenntnisse.
☐ nur theoretische Kenntnisse.
☐ weniger als 300 Programmzeilen selbst geschrieben.
☐ weniger als 3.000 Programmzeilen selbst geschrieben.
☐ weniger als 30.000 Programmzeilen selbst geschrieben.
☐ mehr als 30.000 Programmzeilen selbst geschrieben.

Das größte Programm, das ich bislang *alleine geschrieben* habe, hatte ungefähr ______________ (Anzahl Zeilen) Programmzeilen Quellcode in der Sprache ____________________ (Programmiersprache) und verschlang einen Zeitaufwand von ______________ (Personenmonate) Personenmonaten.

Die folgende Frage ist nur auszufüllen, wenn Sie schon einmal an einem Team-Softwareprojekt mitgearbeitet haben oder das gegenwärtig tun. Das größte Programm, an dem ich *mitgewirkt* habe, hatte insgesamt ungefähr ______________ (Anzahl Zeilen) Programmzeilen und verschlang einen Zeitaufwand von ______________ (Personenmonate) Personenmonaten. Mein eigener Beitrag waren etwa ______________ (Anzahl Zeilen) Programmzeilen bzw. ______________ (Personenmonate) Personenmonate.

Mein Verständnis von Entwurfsmustern ist folgendermaßen: (Tragen sie für jedes Entwurfsmuster eine Ziffer von 1 bis 5 ein, die angibt, wie gut Sie subjektiv glauben, das Entwurfsmuster verstanden zu haben, und zwar:
1: verstehe und beherrsche ich sehr genau,
2: verstehe ich gut,
3: verstehe ich halbwegs,
4: verstehe ich in Ansätzen,
5: verstehe ich nicht)

_____ Abstrakte Fabrik (Abstract Factory).

_____ Befehl (Command).

_____ Beobachter (Observer): Wie `Modell/Observer` in JAKK-Aufgabe 3 („Sichten“)

_____ Besucher (Visitor): Wie z.B. `PrintKomponenten` in JAKK-Aufgabe 2 („Visitor“)

_____ Brücke (Bridge).

_____ Fabrikmethode (Factory Method).

_____ Iterator (Iterator).

_____ Kompositum (Composite): Wie `Kompomente/Behälter` in JAKK-Aufgabe 2 („Visitor“)

_____ Proxy (Proxy).

_____ Schablonenmethode (Template Method): Wie z.B. `suche()` in JAKK-Aufgabe 4 („Verfahren“)

_____ Strategie (Strategy): Wie `ThreadManager/Killer/Waiter` in JAKK-Aufgabe 4 („Verfahren“)

_____ Vermittler (Mediator).

_____ Zuständigkeitskette (Chain of Responsibility): Wie bei Verwaltung der `action()`–Methoden von AWT-Komponenten

Bitte tragen Sie nun Matrikelnummer und Zeit ein und lassen Sie sich die nächsten Unterlagen geben.

Matrikelnummer: Uhrzeit:

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

Fragebogen Teil 2: Entwurfsmuster

a.) Die nachfolgende Tabelle enthält in jeder Spalte ein Entwurfsmuster und in jeder Zeile die Bezeichnung einer Operation, die eine Methode in diesem Entwurfsmuster ausführen könnte. Der tatsächliche Name der Methode in einem Auftreten des Entwurfsmusters wird natürlich in der Regel anders lauten; die Bezeichnung hier gibt nur den allgemeinen Zweck an.

Kreuzen Sie in jeder Zeile alle diejenigen Entwurfsmuster an, in denen die jeweilige Operation in der Regel vorkommt. (Die vollen Namen der abgekürzten Entwurfsmuster sind „Schablonenmethode“ und „Zuständigkeitskette“.)

Operation	Be fehl	Beob achter	Besu cher	Kom posi tum	Schbl. meth ode	Stra tegie	Ver mitt ler	Zust. kette
nimmEntgegen(), accept()								
meldeAn(), register()								
fuehreAus(), execute()								
fuegeZu(), add()								
benachrichtige(), notify()								
aktualisiere(), update()								

Beantworten Sie stichwortartig die folgenden Fragen:

b.) Was ist die Alternative zur Einführung eines Besuchers?

c.) Charakterisieren Sie die Unterschiede zwischen „Schablonenmethode“ und „Strategie“.

d.) Wie vermeidet man bei „Kompositum“ das Problem, dass auch die Blätterklassen diejenigen Operationen haben, die nur für Behälter sinnvoll sind?

Bitte tragen Sie nun Matrikelnummer und Zeit ein und lassen Sie sich die nächsten Unterlagen geben.

Matrikelnummer: Uhrzeit:

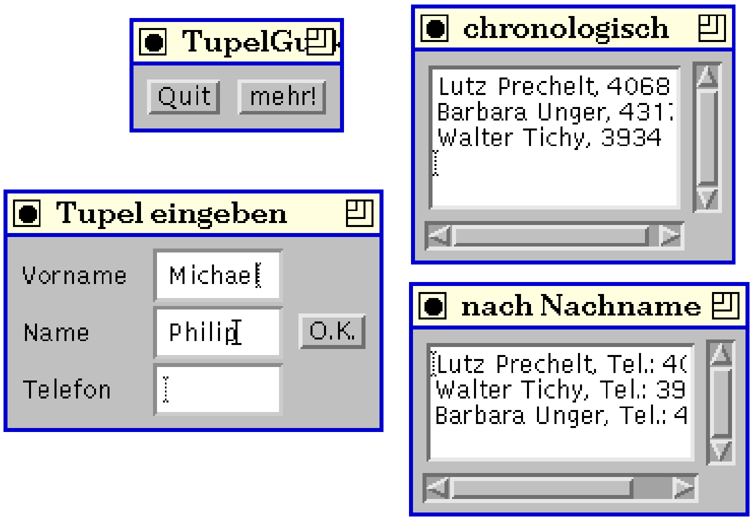
Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

Aufgabe „Tupel“

Sie haben ein Listing des Programms „Tupel“ erhalten.

Das Programm öffnet jeweils bei Betätigen des Knopfes „mehr!“ ein Fenster „Tupel eingeben“, in dem es einen einzelnen Datensatz (Vorname, Nachname, Telefonnummer) vom Benutzer einliest. Alle diese Datensätze werden dann in zwei verschiedenen Formen in anderen Fenstern zusammenfassend angezeigt. Das eine Fenster („chronologisch“) zeigt die Einträge in der Reihenfolge des Eingebens an, das andere („nach Nachname“) nach dem Nachnamen sortiert und in einem etwas anderen Format.

Die Bildschirmansicht ist in folgender Abbildung dargestellt:



Lesen und verstehen Sie dieses Programm so weit, dass sie die unten beschriebenen Arbeitsaufträge daran erledigen können. Wenn das Verständnis gewisser Programmteile Ihnen dafür nicht notwendig erscheint, brauchen Sie diese nicht zu betrachten.

Bitte schreiben Sie die Lösungen nicht in das Programmlisting, sondern *nur* auf die Zettel mit den Aufgabenstellungen. Sie können aber gerne Markierungen und Notizen im Listing machen. Geben Sie in jedem Fall das Listing wieder mit ab.

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

Arbeitsaufträge „Tupel“

1. Welche Programmzeilen müssen geändert werden, um aus den Kommas zwischen Nachname und Telefonnummer im „chronologisch“ Fenster einen Doppelpunkt zu machen?

Zeilennummern

2. Welche Programmzeilen müssen geändert werden, damit das Fenster „nach Nachname“ 40 Zeichen breit dargestellt wird?

Zeilennummern

3. Erweitern Sie das Programm. Erzeugen Sie ein drittes Fenster „auswärts“, in dem, sortiert nach Nachnamen, nur diejenigen Tupel angezeigt werden, deren Telefonnummer mit einer Null (also mit einer Vorwahl) beginnt.

a.) Welche Klasse führen Sie ggf. hierfür neu ein? Welche Methoden müssen implementiert werden?
Geben Sie (z.B. auf der Rückseite) die komplette Klassendefinition ausgenommen die Methodenrümpfe an; also Klassenkopf, Variablendeklarationen und Methodenköpfe.

b.) An welchen Stellen im Programm müssen Sie über die neu eingeführten Klassen und Methoden hinaus einzelne Anweisungen ergänzen? Welche?
Geben Sie Zeilennummern und konkrete Anweisungen an.

Zeilennr. Anweisung(en)

Zeilennr. Anweisung(en)

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

c.) Beschreiben Sie kurz, wie Sie vorgegangen sind, um das nötige Programmverständnis zur Lösung der Teilaufgaben a.) und b.) zu bekommen.

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

4. Erweitern Sie das Programm.

a.) Stellen Sie eine weitere TupelAnzeiger-Klasse TupelAnzR zur Verfügung, die NTTupel im Format
`Nachname, Vorname; Telefonnummer`
in invers chronologischer Reihenfolge anzeigt (also immer das jüngste Tupel zuoberst).
Sie brauchen keine Benutzung der Klasse einzuführen.

Geben Sie die komplette Klassendefinition ausgenommen die Methodenrümpfe an; also Klassenkopf, Variablendeklarationen und Methodenköpfe.

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

b.) Beschreiben Sie kurz, wie Sie vorgegangen sind, um das nötige Programmverständnis zur Lösung von Teilaufgabe a.) zu bekommen.

Bitte tragen Sie nun Matrikelnummer und Zeit ein und lassen Sie sich die nächsten Unterlagen geben.

Matrikelnummer: Uhrzeit:

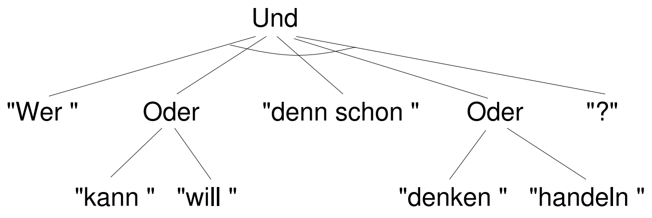
Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

Aufgabe „Element“

Sie haben ein Listing des Programms „Element“ erhalten.

Das Programm enthält eine Bibliothek zum Aufbau von Und/Oder-Bäumen, deren Blätter je eine Zeichenkette (String) enthalten. Wenn man Und als Verkettung und Oder als Alternation interpretiert, definiert ein solcher Baum eine Menge von Zeichenketten. Diese Menge von Zeichenketten wird die *Varianten* des Baums genannt und von der Methode variants() berechnet. Andere Methoden erzeugen eine kompakt kodierte Darstellung des Baums oder berechnen die maximale Tiefe von Und-Knoten, Oder-Knoten und Blättern. Das Hauptprogramm erzeugt einen solchen Baum und gibt seine 4 Varianten, dann seine kompakte Darstellung und schließlich die Tiefeninformation aus.

Der erzeugte Baum hat folgende Form:



Das Programm erzeugt folgende Ausgabe:

```
Wer kann denn schon denken ?
Wer kann denn schon handeln ?
Wer will denn schon denken ?
Wer will denn schon handeln ?
UND("Wer "&ODER("kann "|"will ")&"denn schon "&ODER("denken "|"handeln ")&"?")
  Tiefe=2  UndTiefe=0  OderTiefe=1
```

Lesen und verstehen Sie dieses Programm so weit, dass sie die unten beschriebenen Arbeitsaufträge daran erledigen können. Wenn das Verständnis gewisser Programmteile Ihnen dafür nicht notwendig erscheint, brauchen Sie diese nicht zu betrachten.

Bitte schreiben Sie die Lösungen nicht in das Programmlisting, sondern *nur* auf die Zettel mit den Aufgabenstellungen. Sie können aber gerne Markierungen und Notizen im Listing machen. Geben Sie in jedem Fall das Listing wieder mit ab.

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

Arbeitsaufträge „Element“

1. Welche Programmzeilen müssen geändert werden, um zwischen denjenigen Teilen jeder Variante, die durch Verketten zusammengefügt wurden, stets ein zusätzliches Leerzeichen einzufügen?

Zeilennummern

2. Mit welchem Ausdruck könnte man nach der Erzeugung von u in main() berechnen, *wie viele* Varianten u hat?

Ausdruck

3. Erweitern Sie das Programm. Die obige Art, die Zahl der Varianten zu bestimmen, ist ineffizient, weil dabei eventuell ein recht großes Datenobjekt erzeugt wird, das gar nicht benötigt wird.
Geben Sie deshalb eine Programmerweiterung an, die die Anzahl von Varianten berechnet, *ohne* ein solches großes Datenobjekt zu erzeugen;

a.) Welche Klasse führen Sie ggf. hierfür neu ein? Welche Methoden müssen implementiert werden?
Geben Sie (z.B. auf der Rückseite) die komplette Klassendefinition ausgenommen die Methodenrümpfe an; also Klassenkopf, Variablendeklarationen und Methodenköpfe.

b.) Welche Anweisungen müssen Sie wo ergänzen, um die neue Berechnung zu benutzen und ihr Ergebnis auszugeben? Geben Sie Zeilennummern und konkrete Anweisungen an.

Zeilennr. Anweisung(en)

Zeilennr. Anweisung(en)

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

c.) Beschreiben Sie kurz, wie Sie vorgegangen sind, um das nötige Programmverständnis zur Lösung der Teilaufgaben a.) und b.) zu bekommen.

Bitte tragen Sie nun Matrikelnummer und Zeit ein und lassen Sie sich die nächsten Unterlagen geben.

Matrikelnummer: Uhrzeit:

Bitte tragen Sie beim ersten gründlichen Lesen dieser Seite die Uhrzeit ein. Uhrzeit:

Fragebogen Teil 3: Erfahrungen im Experiment

Dieser Teil des Fragebogens dient dazu, die Antworten auf die Aufgaben durch subjektive Hintergrundinformation zu ergänzen, um eine bessere Analyse des Experiments zu ermöglichen.

Einige Fragen sind gefolgt von etwas Platz für einen Freitextkommentar; dort eingetragene Zusatzinformation ist ggf. sehr wertvoll für uns.

Kreuzen Sie an, welche der folgenden Muster Ihrer Meinung nach in welchen der Programme vorgekommen sind. Wo Sie sich nicht sicher sind, tragen Sie statt eines Kreuzes ein Fragezeichen ein.

Aufgabe	Be fehl	Beob achter	Besu cher	Kom posi tum	Schbl. meth ode	Stra tegie	Ver mitt ler	Zust. kette
Aufgabe „Element“								
Aufgabe „Tupel“								

Kommentar:

Bei den nachfolgenden Fragen stehen vor jeder Antwortmöglichkeit zwei Felder zum Ankreuzen:
[E] für die Aufgabe „Element“ und
[T] für die Aufgabe „Tupel“.
Kreuzen Sie also pro Frage in jeder Antwortspalte genau ein Feld an.

Ich fand die Aufgabe „Element“ bzw. „Tupel“ in der gegebenen Situation insgesamt

[E] [T] recht einfach.
[E] [T] nicht ganz so einfach.
[E] [T] recht schwierig.
[E] [T] schwierig.

Kommentar:

Meine Konzentrationsfähigkeit war bei der Aufgabe „Element“ bzw. „Tupel“ insgesamt

[E] [T] sehr gut.
[E] [T] gut.
[E] [T] ausreichend.
[E] [T] eher schlecht.
[E] [T] schlecht.

Kommentar:

Ich glaube, dass meine Lösungen für die Aufgabe „Element“ bzw. „Tupel“

- E T keine Fehler oder Lücken enthalten.
- E T nur höchstens einen Fehler oder eine Lücke enthalten.
- E T wahrscheinlich noch mehrere Fehler oder Lücken enthalten.
- E T (weiß nicht).

Kommentar:

Ich glaube, dass mir meine Vorkenntnisse über Entwurfsmuster bei der Lösung der Aufgabe „Element“ bzw. „Tupel“ geholfen haben.

- E T Nein, gar nicht.
- E T Nur wenig.
- E T Bin unentschieden.
- E T Ja, etwas.
- E T Ja, sehr.

Kommentar:

Ich glaube, dass mir, soweit vorhanden, die Kennzeichnung von vorhandenen Entwurfsmustern im Programm bei der Lösung der Aufgabe „Element“ bzw. „Tupel“ geholfen hat.

- E T Nein, gar nicht.
- E T Nur wenig.
- E T Bin unentschieden.
- E T Ja, spürbar.
- E T Ja, sehr.

Kommentar:

Ich habe mich zur Lösung von Aufgabe „Element“ bzw. „Tupel“ aktiv auf die Suche nach Entwurfsmustern im Programm gemacht.

- E T Nein, weil es mir gar nicht nötig erschien, die im Programm vorhandenen Entwurfsmuster zu erkennen.
- E T Nein, weil die Entwurfsmuster offensichtlich dokumentiert waren.
- E T Nein, weil ich die Entwurfsmuster gleich entdeckt habe.
- E T Nein, weil

  ____
  Grund bei „Element“

  ____
  Grund bei „Tupel“
- E T Ja, von Anfang an.
- E T Ja, aber erst als ich anders nicht mehr weiter kam.
- E T Ja, aber erst nach einer Weile, denn

  ____
  Grund bei „Element“

  ____
  Grund bei „Tupel“

Kommentar:

Folgendes möchte ich noch loswerden (was ich besonders schwierig fand, ungeschickt am Experimentaufbau finde, interessant am Experiment oder meinem Verhalten darin finde, etc.):

Danke!

Vielen Dank für die Teilnahme an unserem Experiment. Wir hoffen, Sie haben dabei ebensoviel gelernt wie wir. Ihren Praktikumsschein können Sie in der letzten Vorlesungswoche im Sekretariat in Raum 368 Informatikgebäude abholen.

Bitte tragen Sie nun nochmals Matrikelnummer und Zeit ein und geben Sie alle Unterlagen ab.

Matrikelnummer: Uhrzeit:

P.S.: Wenn Sie die Unterlagen abgegeben haben, können Sie entweder ein Buch rausholen und anfangen zu lesen, bis auch die übrigen Teilnehmer mit dem Experiment fertig sind. Das wäre uns am liebsten, weil es die anderen am wenigsten stört.

Oder sie können leise *über die Bank steigen und den Hörsaal verlassen. Bei Diskussionen mit Anderen draußen halten Sie bitte genug Abstand von der Hörsaaltür, um die Weiterarbeitenden nicht zu stören.*

A.2 English translation

Instructions and Questionnaire for the Java/AWT experiment

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

June 16, 1997

Student Id:

(Please do not write here)

Task	Points
E	
S	
Ms	
Ma	
Mb	
Mc	
Md	
T1	
T2	
T3a	
T3b	
T3c	
T4a	
T4b	
E1	
E2	
E3a	
E3b	
E3c	

Questionnaire Part 1: Personal Information

Please fill in this part of the questionnaire before further reading 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 semester,
subject
☐ with Vordiplom, ☐ still without Vordiplom.

☐ I attended the "Software Engineering I" lectures *this semester* and learned about several design patterns there.

Before the Java 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 intensity):

____
language 1, language 2, ...

Before the Java 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 Java 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 and 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 to understand the design pattern.
1: I understand and command it very well,
2: I understand it well,
3: I understand it roughly,
4: I begin to understand it,
5: I do not understand it)

______ Abstract Factory.

______ Command.

______ Observer. E.g `Model/Observer` in course exercise 3 (“Views”)

______ Visitor. E.g. `PrintComponents` in course exercise 2 (“Visitor”)

______ Bridge.

______ Factory Method.

______ Iterator.

______ Composite. E.g. `Compoment/Container` in course exercise 2 (“Visitor”)

______ Proxy.

______ Template Method. E.g. `search()` in course exercise 4 (“Procedure”)

______ Strategy. E.g. `ThreadManager/Killer/Waiter` in course exercise 4 (“Procedure”)

______ Mediator.

______ Chain of Responsibility. E.g. management of `action()` methods in AWT components

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 names of the abbreviated patterns are “Template Method” and “Chain of Responsibility”.)

Operation	Com mand	Obser ver	Visi tor	Com posite	Tem plate Meth.	Stra tegy	Medi ator	Chain of Resp.
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:

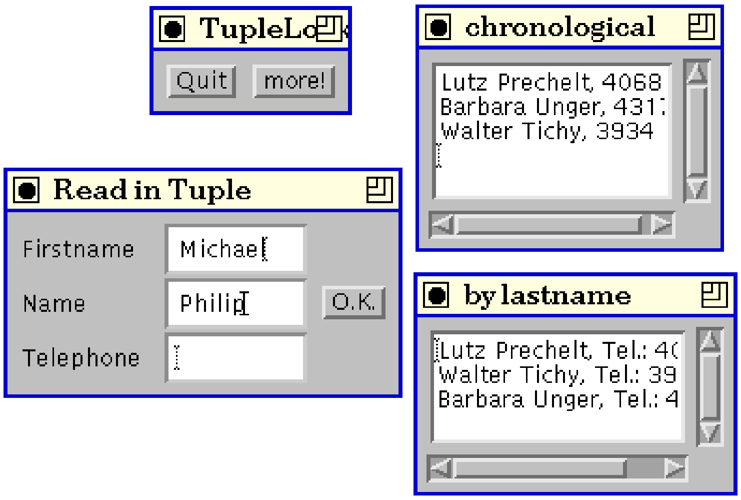
Please enter the time here when you first read this page thoroughly. Time:

Task “Tuple”

You have received a listing of the program “Tuple”.

Whenever the button “more!” is clicked, the program will create a window “Read in Tuple” and read a single record (firstname, lastname, telephone number) typed into that window by the user. All these records are displayed together in other windows in two different styles. One window (“chronological”) shows the records in the order in which they were entered. The other window (“by lastname”) shows them sorted by lastname and in a somewhat different format.

Here is a screenshot:



Read and 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 do not write the solutions into the program listing, write *only* on the task description sheets. You may still put markings and notes in the listing, though. In any case, please return the listing when you are finished.

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

Assignments for “Tuple”

1. Which lines in the code must be changed to change the commas between lastname and phone number into colons in the “chronological” window?

line numbers

2. Which lines in the code must be changed to display the “by lastname” window 40 characters wide?

line numbers

3. Extend the program. Generate a third window “nonlocal” in which (sorted by lastname) only those Tuples are displayed whose phone number starts with a zero (i.e., an area code).

a.) Which new class, if any, do you introduce? Which methods have to be implemented?
Write (e.g. on the back of this page) a complete class definition, except for the bodies of the methods; that is, write a class head, variable definitions, and method heads.

b.) Where in the program do individual statements need to be inserted in addition to the new classes and methods? What statements?
Enter line numbers and concrete statements.

line no. statement(s)

line no. statement(s)

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

c.) Shortly describe how you proceeded to gain the program understanding required to solve parts a.) and b.)

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

4. Extend the program.

a.) Write another TupleDisplay class TupleDispR that displays NTTuple in the format
`lastname, firstname; phone number`
in inversely chronological order (that is, the youngest Tuple at the top).
You need not actually use the class (introduce an instance).

Write a complete class definition, except for the bodies of the methods; that is, write a class head, variable definitions, and method heads.

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

b.) Shortly describe how you proceeded to gain the program understanding required to solve part a.)

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

Student Id: Time:

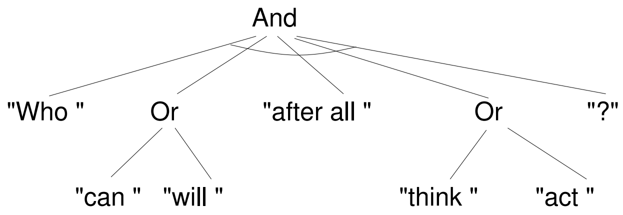
Please enter the time here when you first read this page thoroughly. Time:

Task "Element"

You have received a listing of the program "Element".

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 and finally the depth information.

The tree generated by the program has the following structure:



The program generates the following output:

```
Who can after all think ?
Who can after all act ?
Who will after all think ?
Who will after all act ?
AND("Who "&OR("can "|"will ")&"after all "&OR("think "|"act ")&"?")
  Depth=2  AndDepth=0  OrDepth=1
```

Read and 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 do not write the solutions into the program listing, write *only* on the task description sheets. You may still put markings and notes in the listing, though. In any case, please return the listing when you are finished.

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

Assignments for “Element”

1. Which lines in the code must be changed to insert an additional space character between those parts of each variant that were combined by concatenation?

____
line numbers

2. After u was generated in main(), which expression could be used to compute what the *number of variants* of u is?

____
Expression

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.

a.) Which new class, if any, do you introduce? Which methods have to be implemented?
Write (e.g. on the back of this page) a complete class definition, except for the bodies of the methods; that is, write a class head, variable definitions, and method heads.

b.) Which statements must be inserted in order to use the new computation and print its result? Enter line numbers and concrete statements.

______ ______________________________
line no. statement(s)

______ ______________________________
line no. statement(s)

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

c.) Shortly describe how you proceeded to gain the program understanding required to solve parts a.) and b.)

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 free space below for arbitrary comments; such additional comments may 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.

Aufgabe	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.):

Thank you!

Many thanks for participating in our experiment. We hope you learned as much as we did. You can get your course certificate in room 368 of the informatics building during the last week of the semester.

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

Student Id: Time:

P.S.: After you have returned your materials, you may read a book until the other participants have finished the experiment. We would prefer this, because it causes the least disturbance.

Or you may quietly *leave the room. If you discuss the experiment outsides, please keep away from the door, so not to disturb the students still working.*

Appendix B

Experiment program listings

Consider the possibility that the programmer
did not know what he was doing.
Lionel Deimel & Fernando Naveda, "Reading Computer Programs"

This appendix contains the program listings as given to the subjects during the experiment. Each listing was handed out together with the corresponding task description and had to be given back together with it as well.

The listings are given here in the version with PD. The corresponding versions without PD are exactly the same except that those parts of any comment from the PD marker to the end of the comment are missing. The PD marker is **** ENTWURFSMUSTER: **** in the German program listings and **** DESIGN PATTERN: **** in the English program listings.

The subjects received the listings in the same font (11 point) and line layout as below, but with different page breaks and without header and footer lines (just with plain page numbers bottom-centered).

There are two versions of each listing in this report: The original German version as given to the subjects and an English translation. The latter was kept as close to the original as possible and therefore is somewhat rude at some points.

B.1 Program "Tuple" (German original)

```
/* Tupel
    m1744
    Lutz Prechelt, Barbara Unger
    1996-12-05
    RCS: $Id: TupelMain.java,v 1.2 1997/01/22 16:42:12 prechelt Exp prechelt $
*/
import java.awt.*;
import java.util.Enumeration;
import java.util.Vector;

/*
Dieses Programm verwaltet Mengen von Tupeln.
Ein Tupel besteht aus mehreren Feldern, z.B. Name, Vorname, Telefonnummer.
Es gibt potentiell mehrere solche Tupeltypen und zu jedem davon einen
```

```
Tupelmengentyp.
Das Programm enthaelt ferner GUI-Klassen, mit denen Tupel eines
bestimmten Tupeltyps eingelesen werden koennen und mit denen Tupelmengen
eines bestimmten Tupeltyps angezeigt werden koennen.
Derzeit wird nur der Tupeltyp NTTupel voll unterstuetzt.

Hier die Klassenhierarchie:
  interface Tupel
      class NTTupel
      class NATupel
  class Tupelmenge
  class java.awt.Frame
      class NTEinleser
      class NAEinleser
      abstract class TupelAnzeiger
          class NTTupelAnz1
          abstract class TupelAnzA
              class NTTupelAnz2
      class TupelMain

*** ENTWURFSMUSTER: ***

TupelAnzeiger ist die abstrakte Oberklasse fuer **Beobachter**
der Datenstruktur Tupelmenge.

TupelAnzA ist die abstrakte Oberklasse fuer eine Gruppe solcher
Beobachter, die sich hinsichtlich Auswahl, Anordnung und Formatierung
der darzustellenden Tupel mit Hilfe einer **Schablonenmethode**
variieren lassen.
*/


//--------------- Tupel -------------------
/**
  Gemeinsame Schnittstelle aller Tupeltypen.
*/
interface Tupel {
  /** Liest ein Tupel ein und liefert es bei der angegebenen Tupelmenge
       ab.
  */
  public void getTupel(Tupelmenge m);
}

//--------------- NTTupel -------------------
/**
  * Eine Implementierung von Tupel fuer Namen und Telefonnummern
  */
class NTTupel implements Tupel {
  private String name;                                   // Name einer Person
  private String vorname;                                // Vorname der Person
```

```
    private String telefon;                              // Telefonnummer der Person

    public String getName() { return name; }

    public String getVorname() { return vorname; }

    public String getTelefon() { return telefon; }

    public void setName(String n) { name = n; }

    public void setVorname(String v) { vorname = v; }

    public void setTelefon(String t) { telefon = t; }

    /** Liest ein Tupel ein. Die genaue Strategie hierfuer ist
        in getTupel verkapselt. getTupel garantiert nur,
        dass das Tupel nach dem Einlesen bei der Tupelmenge m gemeldet wird
        und danach auch selbst die richtigen Feldwerte enthaelt.
        getTupel garantiert NICHT, dass das Einlesen bei Rueckkehr aus
        getTupel bereits abgeschlossen ist.
    */
    public void getTupel(Tupelmenge m) {
        NTEinleser e = new NTEinleser(this, m);          // einlesen mit GUI, asynchron
    }
}


//—————— NTEinleser ——————
/**
    GUI-Einleseklasse fuer NTTupel.
    Erst hierin wird endgueltig die Entscheidung getroffen, dass das
    Einlesen mit einem GUI erfolgt.
*/
final class NTEinleser extends Frame {
    private Button ok;
    private TextField t1, t2, t3;
    private Label l1, l2, l3;
    private NTTupel tupel;
    private Tupelmenge m;

    /** Erzeugt Fenster mit drei Textfeldern und einem OK-Knopf zum
        Einlesen der Teilwerte eines NTTupel
    */
    public NTEinleser(NTTupel t, Tupelmenge m) {
        super("Tupel eingeben");
        Panel p = new Panel();
        p.setLayout(new GridLayout(3, 2));
        this.setLayout(new FlowLayout());
        l1 = new Label("Vorname");
        l2 = new Label("Name");
```

```
        l3 = new Label("Telefon");
        t1 = new TextField();
        t2 = new TextField();
        t3 = new TextField();
        ok = new Button("O.K.");
        p.add(l1); p.add(t1);                    // trage Label/Text-Paare in 3 Zeilen in p ein
        p.add(l2); p.add(t2);
        p.add(l3); p.add(t3);
        this.add(p);
        this.add(ok);
        this.tupel = t;                          // wird gebraucht, wenn OK-Knopf gedrueckt wurde.
        this.m = m;                              // dito
        this.pack();                             // berechne layout
        this.show();                             // zeige Dialogfenster an
    }

    /** Ereignisbehandlungsmethode. Verstaut die eingelesenen Werte im
         Tupel, wenn der OK-Knopf gedrueckt wurde und meldet das Tupel
         an die Tupelmenge.
    */
    public boolean action(Event ev, Object target) {
        if (ev.id == Event.ACTION_EVENT) {
            if (ev.target == ok) {               // OK-Knopf gedrueckt
                tupel.setVorname(t1.getText());
                tupel.setName(t2.getText());
                tupel.setTelefon(t3.getText());
                m.neuesTupel(tupel);
                this.hide();                     // entferne Frame von Bildschirm
                this.dispose();                  // gib seine Resourcen frei
                return (true);
            }
            else {
                return (false);
            }
        }
        else {
            return (false);
        }
    }
}


//------------ NATupel ---------------
/**
     Eine Implementierung von Tupel fuer Namen und Adressen
*/
class NATupel implements Tupel {
    private String name;                         // Name einer Person
    private String vorname;                      // Vorname der Person
    private String strasse;                      // Strasse und Hausnummer der Person
```

```
  private String ort;                                  // Postleitzahl und Ort der Person

  public String getName() { return name; }

  public String getVorname() { return vorname; }

  public String getStrasse() { return strasse; }

  public String getOrt() { return ort; }

  public void setName(String n) { name = n; }

  public void setVorname(String v) { vorname = v; }

  public void setStrasse(String s) { strasse = s; }

  public void setOrt(String o) { ort = o; }

  /** Liest ein Tupel ein. Die genaue Strategie hierfuer ist
       in getTupel verkapselt. getTupel garantiert nur,
       dass das Tupel nach dem Einlesen bei der Tupelmenge m gemeldet wird
       und danach auch selbst die richtigen Feldwerte enthaelt.
       getTupel garantiert NICHT, dass das Einlesen bei Rueckkehr aus
       getTupel bereits abgeschlossen ist.
  */
  public void getTupel(Tupelmenge m) {
    NAEinleser e = new NAEinleser(this, m);             // einlesen mit GUI, asynchron
  }
}


//--------------- NAEinleser ---------------
/**
    GUI-Einleseklasse fuer NATupel.
    Erst hierin wird also endgueltig die Entscheidung getroffen, dass das
    Einlesen mit einem GUI erfolgt.
*/
final class NAEinleser extends Frame {
  private Button ok;
  private TextField t1, t2, t3, t4;
  private Label l1, l2, l3, l4;
  private NATupel tupel;
  private Tupelmenge m;

  /** Erzeugt Fenster mit drei Textfeldern und einem OK-Knopf zum
       Einlesen der Teilwerte eines NTTupel
  */
  public NAEinleser(NATupel t, Tupelmenge m) {
    super("Tupel eingeben");
    Panel p = new Panel();
```

```
        p.setLayout(new GridLayout(4, 2));
        this.setLayout(new FlowLayout());
        l1 = new Label("Vorname");
        l2 = new Label("Name");
        l3 = new Label("Strasse");
        l4 = new Label("Ort");
        t1 = new TextField();
        t2 = new TextField();
        t3 = new TextField();
        t4 = new TextField();
        ok = new Button("O.K.");
        p.add(l1); p.add(t1);                 // trage Label/Text-Paare in p ein
        p.add(l2); p.add(t2);
        p.add(l3); p.add(t3);
        p.add(l4); p.add(t4);
        this.add(p);
        this.add(ok);
        this.tupel = t;                       // wird gebraucht, wenn OK-Knopf gedrueckt wurde.
        this.m = m;                           // dito
        this.pack();                          // berechne layout
        this.show();                          // zeige Dialogfenster an
    }

    /** Ereignisbehandlungsmethode. Verstaut die eingelesenen Werte im
        Tupel, wenn der OK-Knopf gedrueckt wurde und meldet das Tupel
        an die Tupelmenge.
    */
    public boolean action(Event ev, Object target) {
        if (ev.id == Event.ACTION_EVENT) {
            if (ev.target == ok) {            // OK-Knopf gedrueckt
                tupel.setVorname(t1.getText());
                tupel.setName(t2.getText());
                tupel.setStrasse(t3.getText());
                tupel.setOrt(t4.getText());
                m.neuesTupel(tupel);
                this.hide();                  // entferne Frame von Bildschirm
                this.dispose();               // gib seine Resourcen frei
                return (true);
            }
            else {
                return (false);
            }
        }
        else {
            return (false);
        }
    }
}
```

```
//———————— Tupelmenge ——————————
/**
    Verwaltet einen ganzen Haufen von Tupeln und erlaubt, einzelne Tupel
    diesem Haufen zuzufuegen.
    Verwaltet ferner einen Haufen von Tupelanzeigern, die
    aufgerufen werden, wenn ein Tupel zugefuegt wird.
*/
class Tupelmenge {
  private Vector tupelanzeigervektor = new Vector();          // eingetragene Anzeiger
  private Vector tupelvektor = new Vector();                                 // die Tupel

  /** wird von einem Tupelerzeuger aufgerufen um ein neues Tupel in die
       Tupelmenge einzubringen
  */
  synchronized void neuesTupel(Tupel t) {
      tupelvektor.addElement((Object)t);
      Enumeration e = tupelanzeigervektor.elements();
      while (e.hasMoreElements())
          ((TupelAnzeiger)e.nextElement()).neuesTupel(t);
  }

  void neuerAnzeiger(TupelAnzeiger ta){
     tupelanzeigervektor.addElement((Object)ta);
     ta.neueTupel(tupelvektor);
  }
}


//———————— TupelAnzeiger ——————————
/**
    TupelAnzeiger stellt mehrere Tupel in einem eigenen Fenster dar.
    Jede Unterklasse legt eine eigene Darstellungsform fest.
    TupelAnzeiger garantieren, dass sie keine Veraenderungen auf den
    Tupel-Objekten durchfuehren (so dass man ihnen 'Originale' anvertrauen
    kann und nicht dauernd Kopien machen muss).
    Ein Programm kann eine ganze Reihe von TupelAnzeiger-Objekten herstellen
    und ihnen die gleichen Tupel uebergeben.
*/
abstract class TupelAnzeiger extends Frame {
  private Vector tupelvektor = new Vector();

  TupelAnzeiger(String name) {
      super(name);
  }

  /** Wird vom Benutzer des Tupelanzeigers aufgerufen, um zu verkuenden,
       dass ein neues Tupel zu den bislang schon dargestellten hinzukommen
       soll.
  */
  abstract synchronized void neuesTupel(Tupel t);
```

```

  /** wird vom Benutzer des Tupelanzeigers aufgerufen, um zu verkuenden,
      dass mehrere neue Tupel zu den bislang schon dargestellten hinzukommen
      sollen.
      Der Aufrufer garantiert, dass sich im Vector ausschliesslich
      'Tupel'-Objekte befinden.
      neueTupel() garantiert, dass es 'neuetupel' nicht aendert.
  */
  abstract synchronized void neueTupel(Vector neuetupel);
}


//—————— NTTupelAnz1 ——————
/**
    Stellt NTTupel-Objekte in der Reihenfolge ihrer Anlieferung
    mit allen Einzelteilen in einem einfachen Format dar.
    Die Benutzung von Tupelobjekten anderer Tupeltypen fuehrt zu
    ClassCastException.
*/
final class NTTupelAnz1 extends TupelAnzeiger {
  private TextArea textarea;

  /** oeffnet neues Fenster mit einem TextArea darin, in dem spaeter die
      Tupel angezeigt werden sollen
  */
  NTTupelAnz1(String name) {
    super(name);
    this.setLayout(new FlowLayout());
    textarea = new TextArea(5, 15);
    this.add(textarea);                        // textarea ist einziges Element im Frame
    this.pack();
    this.show();
  }

  synchronized void neuesTupel(Tupel t) {
    NTTupel nt = (NTTupel)t;
    // neues Tupel einfach hinten an bisherige Darstellung anhaengen:
    textarea.appendText(nt.getVorname() + "  " +
                        nt.getName() + ",  " +
                        nt.getTelefon() + "\n");
  }

  synchronized void neueTupel(Vector neuetupel) {
    Enumeration e = neuetupel.elements();
    while (e.hasMoreElements())
      neuesTupel((Tupel)e.nextElement());
  }
}


```

```
//—————— TupelAnzA ——————
/**
    TupelAnzA stellt Tupelmengen in einer Form dar, die durch Einsetzen
    einer Selektiermethode (Tupel ganz zurueckweisen),
    einer Vergleichsmethode (Tupelreihenfolge bestimmen) und
    einer Formatiermethode (Tupelfelder beliebig auswaehlen und formatieren)
    an viele Zwecke angepasst werden kann.
*/
abstract class TupelAnzA extends TupelAnzeiger {
  /** tupelvektor enthaelt alle darzustellenden Tupel (nach dem Ausfiltern!),
        die Reihenfolge spiegelt die Darstellungsreihenfolge wider.
        tupelstringvektor enthaelt parallel zu tupelvektor die endgueltigen
        Anzeigeformen der Tupel.
        Die Inhalte von tupelstringvektor werden direkt in textarea geschrieben.
        textarea enthaelt die endgueltige Gesamtdarstellung.
  */
  private Vector tupelvektor;                                 // darzustellende Tupel
  private Vector tupelstringvektor;           // dazu korrespondierend: ihre Darstellung
  private TextArea textarea;

  TupelAnzA(String name) {
      super(name);
      tupelvektor = new Vector();
      tupelstringvektor = new Vector();
      this.setLayout(new FlowLayout());
      textarea = new TextArea(4, 16);
      this.add(textarea);                     // textarea ist einziges Element im Frame
      this.pack();
      this.show();
  }

  /** Liefert die gewuenschte Darstellung von Tupel a als String.
        Der String darf Null, eines oder mehrere Newline-Zeichen enthalten.
  */
  abstract String formatiere(Tupel a);

  /** Liefert true, wenn das Tupel dargestellt werden soll und
                false wenn nicht
  */
  abstract boolean selektiere(Tupel a);

  /** Liefert true, wenn Tupel a vor Tupel b dargestellt werden soll und
                false, wenn Tupel b vor Tupel a dargestellt werden soll.
  */
  abstract boolean vergleiche(Tupel a, Tupel b);

  /** realisiert das Zufuegen eines neuen Tupels.
        Es wird zunaechst mittels selektiere() bestimmt, ob das Tupel ueberhaupt
        zugefuegt werden soll, dann wird es ggf. in der Methode sortierenEin()
        mittels vergleiche an die richtige Stelle in der Darstellung
```

```
        gebracht und mittels formatiere() in einen String des gewuenschten
        Darstellungsformats verwandelt.
        *** ENTWURFSMUSTER: ***
        neuesTupel() ist zusammen mit seiner Hilfsmethode sortiereEin() eine
        **Schablonenmethode**, wobei die in den Unterklassen aufgefuellten
        Luecken aus den Methoden selektiere(), formatiere() und vergleiche()
        bestehen.
    */
    synchronized void neuesTupel(Tupel t) {
        Tupel nt = (Tupel)t;
        if (!selektiere(nt))
            return;                    // Dieses Tupel soll nicht angezeigt werden
        sortiereEin(nt);               // tut nt in tupelvektor und tupelstringvektor
        stelleDar();
    }

    /** Analog zu neuesTupel(), jedoch in einer Schleife fuer mehrere Tupel
        zugleich, wobei die Darstellung nur einmal am Ende aufgefrischt wird.
    */
    synchronized void neueTupel(Vector neuetupel) {
        Enumeration e = neuetupel.elements();
        while (e.hasMoreElements()) {
            Tupel nt = (Tupel)e.nextElement();
            if (selektiere(nt))
                sortiereEin(nt);
            else
                ;                                          // ignoriere Tupel
        }
        stelleDar();
    }

    /** Sucht die Stelle im tupelvektor, an die das neue Tupel gehoert
        und fuegt es dort ein. Erzeugt ausserdem die endgueltige Anzeigeform
        fuer das Tupel und fuegt diese in tupelstringvektor ein.
    */
    private void sortiereEin(Tupel nt) {
      int wohin = 0;                  // index, bei dem neues Tupel einzusortieren ist.
      while(wohin < tupelvektor.size() &&
              vergleiche((Tupel)tupelvektor.elementAt(wohin), nt))
        wohin++;
      // nun zeigt 'wohin' die Einfuegeposition an.
      // verschiebe nun die restlichen Elemente um eins nach hinten
      // und zwar sowohl in tupelvektor als auch im tupelstringvektor!
      int sz = tupelvektor.size(),
          k = sz-1;                                        // hinten anfangen
      tupelvektor.setSize(sz+1);
      tupelstringvektor.setSize(sz+1);
      while (k >= wohin) {
          tupelvektor.setElementAt(tupelvektor.elementAt(k), k+1);
          tupelstringvektor.setElementAt(tupelstringvektor.elementAt(k), k+1);
```

```
            k--;
        }
        // nun fuege das neue Tupel in tupelvektor und seine fertig
        // formatierte Form in tupelstringvektor an Position 'wohin' ein:
        tupelvektor.setElementAt(nt, wohin);
        tupelstringvektor.setElementAt(formatiere(nt), wohin);
    }

    /** Stellt den Inhalt von tupelstringvektor neu in textarea dar.
    */
    private void stelleDar() {
        StringBuffer sb = new StringBuffer();
        Enumeration e = tupelstringvektor.elements();
        sb.ensureCapacity(textarea.getText().length() + 100);        // optimiert!!!
        while (e.hasMoreElements())
            sb.append((String)e.nextElement());
        textarea.setText(sb.toString());        // komplett neuer Text fuer textarea
    }
}


//———————— NTTupelAnz2 ——————————
/**
    NTTupelAnz2 stellt NTTupel dar, wobei
    1. Tupel, deren Telefonnummer leer ist, weggelassen werden und
    2. die Tupel nach Nachname sortiert werden
    Die Benutzung von Tupelobjekten anderer Tupeltypen fuehrt zu
    ClassCastException.
    *** ENTWURFSMUSTER: ***
    NTTupelAnz2 vervollstaendigt die **Schablonenmethode** neuesTupel()
    von TupelAnzA
*/
final class NTTupelAnz2 extends TupelAnzA {

    NTTupelAnz2(String name) {
        super(name);
    }

    /** Liefert die gewuenschte Darstellung von Tupel a als einzeiligen String
    */
    String formatiere(Tupel a) {
        NTTupel b = (NTTupel)a;
        return (b.getVorname() + "  " +
                b.getName() + ", Tel.:  " +
                b.getTelefon() + "\n");
    }

    /** Tupel soll nur dargestellt werden, wenn es eine Telefonnummer enthaelt.
    */
    boolean selektiere(Tupel a) {
```

```
        return !((NTTupel)a).getTelefon().equals("");
    }

    /** a soll vor b dargestellt werden, wenn es einen alphabetisch kleineren
          Nachnamen hat.
    */
    boolean vergleiche(Tupel a, Tupel b) {
        return (((NTTupel)a).getName().compareTo(((NTTupel)b).getName()) < 0);
    }
}


//———————— TupelMain ————————
/**
     Hauptprogramm. Erzeugt ein Hauptfenster mit zwei Knoepfen, eine
     Tupelmenge und zwei Tupelanzeiger. Einer der Knoepfe erzeugt jeweils
     ein NTTupel und fuegt es der Tupelmenge zu.
     Es gibt keinen statischen Typschutz zwischen dem tatsaechlichen Tupeltyp
     in der Tupelmenge und dem Tupeltyp, der von den Tupelanzeigern erwartet
     wird.
     *** ENTWURFSMUSTER: ***
     Die beiden Tupelanzeiger werden als Beobachter bei der Tupelmenge
     eingetragen.
*/
public final class TupelMain extends Frame {
    Button quit = new Button("Quit");
    Button get = new Button("mehr!");
    Tupelmenge alleTupel = new Tupelmenge();

    public TupelMain(String name) {
        super(name);
        this.setLayout(new FlowLayout());
        this.add(quit);
        this.add(get);
        this.pack();
        this.show();
    }

    /** Ereignisbehandlung fuer das gesamte Programm.
    */
    public boolean action(Event ev, Object target) {
      if (ev.id == Event.ACTION_EVENT) {
        if (ev.target == quit) {                        // Quit-Knopf gedrueckt
            this.hide();                                // mache Hauptfenster unsichtbar
            System.exit(0);                             // Verlasse das Programm
        }
        else if (ev.target == get) {                    // "mehr!"-Knopf gedrueckt
            Tupel t = new NTTupel();
            t.getTupel(alleTupel);
        }
```

```
        else {
          System.out.println ("ACTION_EVENT: " + ev.arg.toString());                //!!!
          return (false);
        }
      }
      else {
        System.out.println ("Event:   " + ev.arg.toString() + "  " + ev.id);        //!!!
        return (false);
      }
      return (true);                // all but the 'return false;' cases have been handled
    }

    /* ——— m a i n : ——— */
    public static void main(String args[]) {
       TupelMain mainwindow = new TupelMain("TupelGucker");
       TupelAnzeiger anz1 = new NTTupelAnz1("chronologisch");
       TupelAnzeiger anz2 = new NTTupelAnz2("nach Nachname");
       mainwindow.alleTupel.neuerAnzeiger(anz1);
       mainwindow.alleTupel.neuerAnzeiger(anz2);
    }
}

```

B.2 Program “Tuple” (English translation)

```
/* Tuple
   m1744
   Lutz Prechelt, Barbara Unger
   1996-12-05
   RCS: $Id: TupleMain.java,v 1.2 1997/01/22 16:42:12 prechelt Exp prechelt $
*/
import java.awt.*;
import java.util.Enumeration;
import java.util.Vector;

/*
This program manages sets of tuples.
A tuple consists of several fields, e.g. name, firstname, telephone number.
There are potentially several such tuple types and for each of them
there is also a tuple set type.
Furthermore, the program contains GUI classes for reading tuples of a
particular type and for displaying tuple sets of a particular tuple type.
Currently only the tuple type NTTuple is fully supported.

This is the class hierarchy:
  interface Tuple
     class NTTuple
     class NATuple
  class Tupleset
```

```
  class java.awt.Frame
      class NTReader
      class NAReader
      abstract class TupleDisplay
          class NTTupleDisp1
          abstract class TupleDispA
              class NTTupleDisp2
      class TupleMain

*** DESIGN PATTERN: ***

TupleDisplay is the abstract Superclass of **Observer**
for the data structure Tupleset.

TupleDispA 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 and use a **Template Method** to vary
these aspects.
*/


//—————————— Tuple ——————————
/**
  Common interface of all tuple types.
*/
interface Tuple {
  /** Reads a tuple and delivers it to the given Tupleset.
  */
  public void getTuple(Tupleset m);
}

//—————————— NTTuple ——————————
/**
  * An implementation of Tuple for names and telephone numbers.
  */
class NTTuple implements Tuple {
  private String name;                       // Name einer Person
  private String firstname;                  // Firstname der Person
  private String telephone;                  // Telephonenumber der Person

  public String getName() { return name; }

  public String getFirstname() { return firstname; }

  public String getTelephone() { return telephone; }

  public void setName(String n) { name = n; }

  public void setFirstname(String v) { firstname = v; }

```

```
  public void setTelephone(String t) { telephone = t; }

  /** Reads a Tuple. The exact strategy used is encapsulated
      in getTuple. getTuple only guarantees, that the Tuple
      will be announced to Tupleset s after reading and
      that its fields contain the correct values afterwards.
      getTuple does NOT guarantee that reading is already
      done when getTuple returns.
  */
  public void getTuple(Tupleset s) {
    NTReader e = new NTReader(this, s);          // read in with GUI, asynchronously
  }
}


//————————— NTReader ——————————
/**
    GUI reader class for NTTuple.
    Only here the final decision of using a GUI for reading is being made.
*/
final class NTReader extends Frame {
  private Button ok;
  private TextField t1, t2, t3;
  private Label l1, l2, l3;
  private NTTuple tuple;
  private Tupleset s;

  /** Generates window with three text fieldss and an OK button
       for reading the element values of an NTTuple.
  */
  public NTReader(NTTuple t, Tupleset s) {
    super("Read in Tuple");
    Panel p = new Panel();
    p.setLayout(new GridLayout(3, 2));
    this.setLayout(new FlowLayout());
    l1 = new Label("Firstname");
    l2 = new Label("Name");
    l3 = new Label("Telephone");
    t1 = new TextField();
    t2 = new TextField();
    t3 = new TextField();
    ok = new Button("O.K.");
    p.add(l1); p.add(t1);                        // enter label/text pairs into p in 3 lines
    p.add(l2); p.add(t2);
    p.add(l3); p.add(t3);
    this.add(p);
    this.add(ok);
    this.tuple = t;                              // is needed when OK button was pressed
    this.s = s;                                  // ditto
    this.pack();                                 // compute layout
```

```
        this.show();                                  // show dialog window
    }

    /** Event handling method. Puts values read into Tuple when OK button
        was pressed and announces Tuple to Tupleset.
    */
    public boolean action(Event ev, Object target) {
        if (ev.id == Event.ACTION_EVENT) {
            if (ev.target == ok) {                    // OK button pressed
                tuple.setFirstname(t1.getText());
                tuple.setName(t2.getText());
                tuple.setTelephone(t3.getText());
                s.newTuple(tuple);
                this.hide();                          // remove Frame from screen
                this.dispose();                       // free its resources
                return (true);
            }
            else {
                return (false);
            }
        }
        else {
            return (false);
        }
    }
}


//————————— NATuple ——————————
/**
    An implementation of Tuple for names and addresses
*/
class NATuple implements Tuple {
    private String name;                  // Name einer Person
    private String firstname;             // Firstname der Person
    private String street;                // Street und Hausnumber der Person
    private String city;                  // Postleitzahl und City der Person

    public String getName() { return name; }

    public String getFirstname() { return firstname; }

    public String getStreet() { return street; }

    public String getCity() { return city; }

    public void setName(String n) { name = n; }

    public void setFirstname(String f) { firstname = f; }

```

```
    public void setStreet(String s) { street = s; }

    public void setCity(String c) { city = c; }

    /** Reads a Tuple. The exact strategy used is encapsulated
          in getTuple. getTuple only guarantees, that the Tuple
          will be announced to Tupleset s after reading and
          that its fields contain the correct values afterwards.
          getTuple does NOT guarantee that reading is already
          done when getTuple returns.
    */
    public void getTuple(Tupleset s) {
        NAReader e = new NAReader(this, s);          // read in with GUI, asynchronously
    }
}


//———————— NAReader ——————————
/**
      GUI reader class for NTTuple.
      Only here the final decision of using a GUI for reading is being made.
*/
final class NAReader extends Frame {
    private Button ok;
    private TextField t1, t2, t3, t4;
    private Label l1, l2, l3, l4;
    private NATuple tuple;
    private Tupleset s;

    /** Generates window with three text fieldss and an OK button
          for reading the element values of an NTTuple.
    */
    public NAReader(NATuple t, Tupleset s) {
        super("Read in Tuple");
        Panel p = new Panel();
        p.setLayout(new GridLayout(4, 2));
        this.setLayout(new FlowLayout());
        l1 = new Label("Firstname");
        l2 = new Label("Name");
        l3 = new Label("Street");
        l4 = new Label("City");
        t1 = new TextField();
        t2 = new TextField();
        t3 = new TextField();
        t4 = new TextField();
        ok = new Button("O.K.");
        p.add(l1); p.add(t1);                        // enter label/text pairs into p
        p.add(l2); p.add(t2);
        p.add(l3); p.add(t3);
        p.add(l4); p.add(t4);
```

```
        this.add(p);
        this.add(ok);
        this.tuple = t;                  // is needed when OK button was pressed
        this.s = s;                      // ditto
        this.pack();                     // compute layout
        this.show();                     // show dialog window
    }

    /** Event handling method. Puts values read into Tuple when OK button
          was pressed and announces Tuple to Tupleset.
    */
    public boolean action(Event ev, Object target) {
        if (ev.id == Event.ACTION_EVENT) {
            if (ev.target == ok) {                     // OK button pressed
                tuple.setFirstname(t1.getText());
                tuple.setName(t2.getText());
                tuple.setStreet(t3.getText());
                tuple.setCity(t4.getText());
                s.newTuple(tuple);
                this.hide();                           // remove Frame from screen
                this.dispose();                        // free its resources
                return (true);
            }
            else {
                return (false);
            }
        }
        else {
            return (false);
        }
    }
}


//——————— Tupleset —————————
/**
        Manages a lot of Tuples and allows to add individual Tuples to
        this lot.
        Furthermore, manages a lot of Tupledisplays, which are called
        when a Tuple is being added.
*/
class Tupleset {
    private Vector tupledisplayvector = new Vector();      // registered Displays
    private Vector tuplevector = new Vector();             // the Tuples

    /** called by a Tuple generator in order to bring a new Tuple
          into the Tupleset.
    */
    synchronized void newTuple(Tuple t) {
        tuplevector.addElement((Object)t);
```

```
        Enumeration e = tupledisplayvector.elements();
        while (e.hasMoreElements())
            ((TupleDisplay)e.nextElement()).newTuple(t);
    }

    void newDisplay(TupleDisplay td){
        tupledisplayvector.addElement((Object)td);
        td.newTuples(tuplevector);
    }
}


//—————— TupleDisplay ——————
/**
    TupleDisplay shows multiple Tuples in a separate window.
    Each subclass defines its own presentation style.
    TupleDisplays guarantee that they do not modify the Tuple objects
    (so that it is possible to commit originals to the TupleDisplay’s
    custody instead of creating copies).
    A program may create a number of TupleDisplay objects and give the
    same Tuple objects to each of them.
*/
abstract class TupleDisplay extends Frame {
    private Vector tuplevector = new Vector();

    TupleDisplay(String name) {
        super(name);
    }

    /** Is called by the user of the TupleDisplay to announce
        that a new Tuple shall be added to the ones already displayed.
    */
    abstract synchronized void newTuple(Tuple t);

    /** Is called by the user of the TupleDisplay to announce
        that multiple new Tuples shall be added to the ones already displayed.
        The caller guarantees that all objects in the Vector are
        Tuple objects.
        newTuples() guarantees that it does not change ’newtuples’
    */
    abstract synchronized void newTuples(Vector newtuples);
}


//—————— NTTupleDisp1 ——————
/**
    Displays NTTuple objects in the order in which they are delivered,
    showing all their components in a simple format.
    Using Tuple objects of other Tuple types results in
    ClassCastException.
```

```
*/
final class NTTupleDisp1 extends TupleDisplay {
  private TextArea textarea;

  /** opens new window with a textarea in it, where Tuples will be
       displayed later on.
  */
  NTTupleDisp1(String name) {
      super(name);
      this.setLayout(new FlowLayout());
      textarea = new TextArea(5, 15);
      this.add(textarea);                    // textarea is the only element in Frame
      this.pack();
      this.show();
  }

  synchronized void newTuple(Tuple t) {
      NTTuple nt = (NTTuple)t;
      // just append new Tuple to current display:
      textarea.appendText(nt.getFirstname() + " " +
                          nt.getName() + ", " +
                          nt.getTelephone() + "\n");
  }

  synchronized void newTuples(Vector newtuples) {
      Enumeration e = newtuples.elements();
      while (e.hasMoreElements())
          newTuple((Tuple)e.nextElement());
  }
}


//————————— TupleDispA —————————
/**
    TupleDispA 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).
*/
abstract class TupleDispA extends TupleDisplay {
  /** tuplevector contains all Tuples that are to be displayed (after
       filtering!), their order represents the presentation order.
       tuplestringvector parallels tuplevector and contains the
       final display presentation of the Tuples.
       The contents of tuplestringvector are written into textarea
       as is. textarea contains the final display contents.
  */
  private Vector tuplevector;                    // Tuples to display
  private Vector tuplestringvector;      // corresponding: their representation
```

```
  private TextArea textarea;

  TupleDispA(String name) {
    super(name);
    tuplevector = new Vector();
    tuplestringvector = new Vector();
    this.setLayout(new FlowLayout());
    textarea = new TextArea(4, 16);
    this.add(textarea);                          // textarea is the only element in Frame
    this.pack();
    this.show();
  }

  /** Returns desired representation of Tuple a as a String.
      The String may contain zero, one, or more newline characters.
  */
  abstract String format(Tuple a);

  /** returns true, if the Tuple should be displayed and
                false otherwise
  */
  abstract boolean select(Tuple a);

  /** returns true, if Tuple a should be displayed before Tuple b and
                false, if Tuple b should be displayed before Tuple a.
  */
  abstract boolean compare(Tuple a, Tuple b);

  /** implements adding a new Tuple.
      First select() is used to test, whether the Tuple should be added
      at all, then mergeIn() moves it to the right place in the
      presentation using compare and format() converts it into a String
      of the desired display format.
      *** DESIGN PATTERN: ***
      newTuple() together with its auxiliary method mergeIn() forms a
      **Template Method**. The empty spots that are filled in subclasses
      are the methods select(), format(), and compare().
  */
  synchronized void newTuple(Tuple t) {
    Tuple nt = (Tuple)t;
    if (!select(nt))
      return;                                    // this Tuple shall not be displayed
    mergeIn(nt);                     // puts nt into tuplevector and tuplestringvector
    display();
  }

  /** Analog to newTuple(), but uses a loop for adding several Tuples
      at once. The actual display is updated only once at the end.
  */
  synchronized void newTuples(Vector newtuples) {
```

```
        Enumeration e = newtuples.elements();
        while (e.hasMoreElements()) {
            Tuple nt = (Tuple)e.nextElement();
            if (select(nt))
                mergeIn(nt);
            else
                ;                                   // ignore Tuple
        }
        display();
    }

    /** Finds the place in tuplevector where the new Tuple should be
        and inserts it there. Generates the final display representation
        of the Tuple and inserts that in tuplestringvector.
    */
    private void mergeIn(Tuple nt) {
      int where = 0;                  // index at which new Tuple has to be inserted.
      while(where < tuplevector.size() &&
              compare((Tuple)tuplevector.elementAt(where), nt))
        where++;
      // now 'where' indicates the target index.
      // move all other elements one position towards the end in
      // both tuplevector and tuplestringvector!
      int sz = tuplevector.size(),
          k = sz-1;                                 // start at end
      tuplevector.setSize(sz+1);
      tuplestringvector.setSize(sz+1);
      while (k ≥ where) {
          tuplevector.setElementAt(tuplevector.elementAt(k), k+1);
          tuplestringvector.setElementAt(tuplestringvector.elementAt(k), k+1);
          k--;
      }
      // now insert new Tuple into tuplevector and its formatted counterpart
      // into tuplestringvector at position 'where':
      tuplevector.setElementAt(nt, where);
      tuplestringvector.setElementAt(format(nt), where);
    }

    /** Displays the contents of tuplestringvector in textarea.
    */
    private void display() {
        StringBuffer sb = new StringBuffer();
        Enumeration e = tuplestringvector.elements();
        sb.ensureCapacity(textarea.getText().length() + 100);      // optimized!!!
        while (e.hasMoreElements())
            sb.append((String)e.nextElement());
        textarea.setText(sb.toString());            // completely new text for textarea
    }
}
```

```

//——————— NTTupleDisp2 —————————
/**
    NTTupleDisp2 displays NTTuple, where
    1. Tuples with an empty telephone number are left out and
    2. Tuples are sorted by (last)name
    Using Tuple objects of other Tuple types results in
    ClassCastException.
    *** DESIGN PATTERN: ***
    NTTupleDisp2 completes the **Template Method** newTuple()
    of TupleDispA
*/
final class NTTupleDisp2 extends TupleDispA {

  NTTupleDisp2(String name) {
    super(name);
  }

  /** returns the desired representation of Tuple a as a single line String.
  */
  String format(Tuple a) {
    NTTuple b = (NTTuple)a;
    return (b.getFirstname() + " " +
            b.getName() + ", Tel.:  " +
            b.getTelephone() + "\n");
  }

  /** Tuple shall be displayed only if it contains a telephone number.
  */
  boolean select(Tuple a) {
    return !((NTTuple)a).getTelephone().equals("");
  }

  /** a shall be displayed before b, if its 'name' is alphabetically smaller.
  */
  boolean compare(Tuple a, Tuple b) {
    return (((NTTuple)a).getName().compareTo(((NTTuple)b).getName()) < 0);
  }
}


//——————— TupleMain —————————
/**
    Main program. Generates a main window with two buttons, one
    Tupleset and two TupleDisplays. One of the buttons creates an
    NTTuple and adds it to the Tupleset.
    There is no static type safety between the actual Tuple type
    stored in the Tupleset and the Tuple type that is expected by
    the TupleDisplays.
    *** DESIGN PATTERN: ***
```

```
     The two TupleDisplays are registered as observers at the Tupleset.
*/
public final class TupleMain extends Frame {
    Button quit = new Button("Quit");
    Button get = new Button("more!");
    Tupleset allTuple = new Tupleset();

    public TupleMain(String name) {
        super(name);
        this.setLayout(new FlowLayout());
        this.add(quit);
        this.add(get);
        this.pack();
        this.show();
    }

    /** Event handling for the whole program.
    */
    public boolean action(Event ev, Object target) {
      if (ev.id == Event.ACTION_EVENT) {
        if (ev.target == quit) {                                // Quit button pressed
            this.hide();                                  // make main window invisible
            System.exit(0);                                            // exit program
        }
        else if (ev.target == get) {                          // "more!" button pressed
            Tuple t = new NTTuple();
            t.getTuple(allTuple);
        }
        else {
          System.out.println ("ACTION_EVENT: " + ev.arg.toString());              //!!!
          return (false);
        }
      }
      else {
        System.out.println ("Event:   " + ev.arg.toString() + "  " + ev.id);       //!!!
        return (false);
      }
      return (true);                   // all but the 'return false;' cases have been handled
    }

    /* ——— m a i n : ——— */
    public static void main(String args[]) {
        TupleMain mainwindow = new TupleMain("TupleLooker");
        TupleDisplay disp1 = new NTTupleDisp1("chronological");
        TupleDisplay disp2 = new NTTupleDisp2("by lastname");
        mainwindow.allTuple.newDisplay(disp1);
        mainwindow.allTuple.newDisplay(disp2);
    }
}
```

B.3 Program “Element” (German original)

```
/* Element
    m2238
    Lutz Prechelt
    1997-01-02
    RCS: $Id: ElementMain.java,v 1.2 1997/01/22 16:42:38 prechelt Exp prechelt $
*/
import java.util.Enumeration;
import java.util.Vector;

/*
Dieses Programm verwaltet Und/Oder-Sequenzen (And/Or-Sequenzen) von Strings.
Solche Sequenzen bestehen aus 'Element'-Objekten. Es gibt drei Arten davon:
- StringElement enthaelt einen einzelnen String.
- AndElement enthaelt eine Liste von Elementen, die in der gegebenen
  Reihenfolge verkettet sind.
- OrElement enthaelt eine Menge von Elementen, von denen je
  eines alternativ eingesetzt benutzt kann.
AndElement- und OrElement-Objekte koennen beliebig verschachtelt werden.
Die unterste Ebene (Blaetter) eines dabei entstehenden Objektbaums wird immer
von StringElement-Objekten gebildet.
OrElemente fuehren zu Varianten. Treffen mehrere OrElemente zusammen,
wird das Kreuzprodukt ihrer Varianten gebildet.

Hier die Klassenhierarchie:
  abstract class Element
      class StringElement
      class AndElement
      class OrElement
  interface ElementAction
      class Depth
  class ElementMain

Die letzte Klasse enthaelt ein Hauptprogramm, das eine And/Or-Sequenz
erzeugt und ausgibt.

*** ENTWURFSMUSTER: ***

Element ist die abstrakte Oberklasse eines **Kompositum**.
StringElement ist der zugehoerige Einzelteiltyp (Blatt),
AndElement und OrElement sind zugehoerige Behaeltertypen (Kompositum).

ElementAction ist die abstrakte Oberklasse eines **Besuchers**.
Die besuchte Datenstruktur sind verschachtelte Element-Behaelter.
*/


//————————— Element —————————
/**
```

```
  Schnittstelle der 'Element'-Klassen StringElement, AndElement, OrElement.
  *** ENTWURFSMUSTER: ***
  Element ist die Oberklasse in einem Kompositum-Muster.
*/
abstract class Element {
  /** Zufuegeoperation. Fuegt, falls moeglich, diesem Element ein weiteres
      Element zu.
  */
  abstract public void add(Element e);

  /** Repraesentationsoperation. Liefert eine kodierte Darstellung des
      Elements als String.
  */
  abstract public String asString();

  /** Ausgabeoperation. Gibt das gesamte Element auf System.out aus.
      Jede Variante erscheint auf einer eigenen Zeile.
  */
  public void print() {
    Vector v = variants();
    Enumeration e = v.elements();
    while (e.hasMoreElements())
      System.out.println((String)e.nextElement());
  }

  /** Repraesentationsoperation. Liefert einen Vektor, in dem jedes Element
      die Stringrepraesentation einer Variante der And/Or-Sequenz enthaelt.
      Jede Variante ist genau einmal im Vektor vertreten.
  */
  abstract public Vector variants();

  /** Verzweigungsoperation. Ruft die zur jeweiligen Unterklasse gehoerige
      Operation von ElementAction auf.
      *** ENTWURFSMUSTER: ***
      Dies ist die 'double dispatch' Prozedur in der besuchten Datenstruktur
      fuer das Besucher-Muster.
  */
  abstract void perform(ElementAction a);
}


//—————————— StringElement ——————————
/**
  Elementklasse, die genau einen String enthaelt.
  *** ENTWURFSMUSTER: ***
  StringElement ist die (einzige) Blattklasse in einem Kompositum-Muster.
*/
class StringElement extends Element {
  String inhalt;                                   // Inhalt des StringElements

```

```
    /** Erzeugt ein StringElement mit leerem Inhalt.
    */
    public StringElement() {
        inhalt = new String();
    }

    /** Erzeugt ein StringElement mit dem als Argument gegebenen Inhalt.
    */
    public StringElement(String s) {
        inhalt = new String(s);
    }

    /** Zufuegeoperation. Fuegt e an bisherigen Stringinhalt an,
         falls e ein Stringelement ist.
         Wirft andernfalls CastException.
    */
    public void add(Element e) {
        StringElement s = (StringElement)e;                    // wirft evtl. CastException
        inhalt = inhalt + s.inhalt;
    }

    /** Repraesentationsoperation. Liefert eine kodierte Darstellung des
         Elements als String in der Form
         "inhalt"
    */
    public String asString() {
        return ("\"" + inhalt + "\"");
    }

    /** Liefert einen Vektor mit genau einem Element, naemlich dem String,
         den das StringElement enthaelt.
    */
    public Vector variants() {
        Vector v = new Vector();
        v.addElement((Object)inhalt);
        return v;
    }

    /** Ruft die zu der uebergebenen ElementAction gehoerende stringAction auf.
         *** ENTWURFSMUSTER: ***
         perform() ist die Verzweigungsoperation fuer Besucher (ElementAction).
    */
    void perform(ElementAction a) {
        a.stringAction(this);
    }
}


//————————— AndElement —————————
/**
```

```
  Und-Sequenz von Elementen.
  (Die Elemente sind also miteinander zu verketten.)
  *** ENTWURFSMUSTER: ***
  AndElement ist eine der Behaelterklassen in einem Kompositum-Muster.
*/
class AndElement extends Element {
  Vector elems;                  // Vektor der Elemente, die zusammen das AndElement bilden

  /** Erzeugt ein AndElement ohne Inhalt.
  */
  public AndElement() {
    elems = new Vector();
  }

  /** Zufuegeoperation. Fuegt e als neues Element an bisherige Sequenz von
      Elementen an.
  */
  public void add(Element e) {
    elems.addElement(e);
  }

  /** Repraesentationsoperation. Liefert eine kodierte Darstellung des
      Elements als String in der Form
      UND(el1&el2&el3)
  */
  public String asString() {
    StringBuffer b = new StringBuffer("UND(");
    Enumeration n = elems.elements();
    while (n.hasMoreElements()) {
      Element e = (Element)n.nextElement();
      b.append(e.asString());
      b.append(n.hasMoreElements() ? "&" : ")");
    }
    return b.toString();
  }

  /** Liefert das Kreuzprodukt saemtlicher Varianten saemtlicher Elemente
      des AndElements. Die Produktbildung geschieht durch Stringverkettung der
      Varianten.
  */
  public Vector variants() {
    Vector result = new Vector();
    Enumeration n = elems.elements();
    while (n.hasMoreElements()) {
      Element e = (Element)n.nextElement();
      Vector 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
            }
            Vector oldresult = result;
            int oldN = oldresult.size();
            int nextN = nextpart.size();
            result = new Vector(oldN * nextN);
            // erzeuge Kreuzprodukt (in Form von Stringverkettung):
            // result := oldresult x nextpart
            for (int oldI = 0; oldI < oldN; oldI++) {
                String currentold = (String)oldresult.elementAt(oldI);
                for (int nextI = 0; nextI < nextN; nextI++) {
                    String currentnext = (String)nextpart.elementAt(nextI);
                    result.addElement(currentold + currentnext);          // Stringverkettung
                }
            }
        }
        return result;
    }

    /** Ruft die zu der uebergebenen ElementAction gehoerende andAction auf.
         *** ENTWURFSMUSTER: ***
         perform() ist die Verzweigungsoperation fuer Besucher (ElementAction).
    */
    void perform(ElementAction a) {
        a.andAction(this);
    }
}


//------------------ OrElement ---------------------
/**
  Oder-Menge von Elementen.
  (Die Elemente sind also alternativ.)
  *** ENTWURFSMUSTER: ***
  OrElement ist eine der Behaelterklassen in einem Kompositum-Muster.
*/
class OrElement extends Element {
  Vector elems;                  // Vektor der Elemente, die zusammen das OderElement bilden

  /** Erzeugt ein OderElement ohne Inhalt.
  */
  public OrElement() {
      elems = new Vector();
  }

  /** Zufuegeoperation. Fuegt e als neues Element in bisherige Menge von
        alternativen Elementen ein.
  */
  public void add(Element e) {
      elems.addElement(e);
```

```
    }

    /** Repraesentationsoperation. Liefert eine kodierte Darstellung des
         Elements als String in der Form
         ODER(el1|el2|el3)
    */
    public String asString() {
        StringBuffer b = new StringBuffer("ODER(");
        Enumeration n = elems.elements();
        while (n.hasMoreElements()) {
            Element e = (Element)n.nextElement();
            b.append(e.asString());
            b.append(n.hasMoreElements() ? "|" : ")");
        }
        return b.toString();
    }

    /** Liefert die Vereinigungsmenge aller Varianten aller Elemente
         des OderElements.
    */
    public Vector variants() {
        Vector result = new Vector();
        Enumeration n = elems.elements();
        while (n.hasMoreElements()) {
            Element e = (Element)n.nextElement();
            Vector nextpart = e.variants();
            Enumeration nn = nextpart.elements();
            result.ensureCapacity(result.size() + nextpart.size());
            while (nn.hasMoreElements())
                result.addElement(nn.nextElement());
        }
        return result;
    }

    /** Ruft die zu der uebergebenen ElementAction gehoerende orAction auf.
         *** ENTWURFSMUSTER: ***
         perform() ist die Verzweigungsoperation fuer Besucher (ElementAction).
    */
    void perform(ElementAction a) {
        a.orAction(this);
    }
}


//—————— ElementAction ———————
/**
    Schnittstelle fuer Operationen auf ElementStrukturen, bei denen die
    einzelnen Teile je nach ihrem Typ unterschiedlich behandelt werden sollen.
    *** ENTWURFSMUSTER: ***
    Entwurfsmuster 'Besucher' (Visitor): ElementAction ist die Oberklasse aller
```

```
   Besucher fuer 'Element'-Kompositumdatenstrukturen.
*/
interface ElementAction {
   void stringAction(StringElement e);
   void andAction(AndElement e);
   void orAction(OrElement e);
}


//--------------- Depth ------------------
/**
   Klasse zum Berechnen von Maximaltiefen der verschiedenen Knotenarten in
   einem And/Or-Elementbaum.
   *** ENTWURFSMUSTER: ***
   Depth ist ein Besucher fuer die Kompositum-Datenstruktur Element.
   Element.perform() ist die Verzweigungsoperation fuer den Besucher.
*/
class Depth implements ElementAction {
   public int maxDepth = -1;                  // Tiefe des tiefsten Elements
   public int maxAndDepth = -1;               // Tiefe des tiefsten AndElements
   public int maxOrDepth = -1;                // Tiefe des tiefsten OrElements
   private int depth = 0;                     // aktuelle Tiefe (Wurzel == 0)

   /** Durchwandert das Element e, um sich zu initialisieren.
         Die Komponenten maxDepth, maxAndDepth und maxOrDepth koennen
         anschliessend abgefragt werden.
   */
   public Depth(Element e) {
      e.perform(this);
   }

   public void stringAction(StringElement e) {
      if (maxDepth < depth)
         maxDepth = depth;
   }

   public void andAction(AndElement u) {
      if (maxAndDepth < depth) {
         maxAndDepth = depth;
         if (maxDepth < depth)
            maxDepth = depth;
      }
      iterate(u.elems.elements());
   }


   public void orAction(OrElement o) {
      if (maxOrDepth < depth) {
         maxOrDepth = depth;
         if (maxDepth < depth)
```

```
            maxDepth = depth;
        }
        iterate(o.elems.elements());
    }

    /** Durchlaufe die enthaltenen Elemente eines AndElement oder OrElement.
    */
    private void iterate(Enumeration n) {
        depth++;                        // enthaltene Elemente liegen eine Ebene tiefer.
        while (n.hasMoreElements()) {
            Element e = (Element)n.nextElement();
            e.perform(this);
        }
        depth--;
    }
}


//———————— ElementMain ————————
/**
    Hauptprogramm. Erzeugt eine And/Or-Sequenz und gibt sie aus.
*/
public class ElementMain {
    public static void main (String args[]) {
        AndElement u = new AndElement();
        OrElement modal = new OrElement();
        OrElement verb = new OrElement();
        u.add(new StringElement("Wer "));
        modal.add(new StringElement("kann "));
        modal.add(new StringElement("will "));
        u.add(modal);
        u.add(new StringElement("denn schon "));
        verb.add(new StringElement("denken "));
        verb.add(new StringElement("handeln "));
        u.add(verb);
        u.add(new StringElement("?"));

        u.print();
        System.out.println(u.asString());
        Depth t = new Depth(u);
        System.out.println(" Tiefe=" + t.maxDepth +
                           " UndTiefe=" + t.maxAndDepth +
                           " OderTiefe=" + t.maxOrDepth);
    }
}

```

B.4 Program “Element” (English translation)

```
/* Element
    m2238
    Lutz Prechelt
    1997-01-02
    RCS: $Id: ElementMain.java,v 1.2 1997/01/22 16:42:38 prechelt Exp prechelt $
*/
import java.util.Enumeration;
import java.util.Vector;

/*
This program manages And/Or-Sequences of Strings.
Such sequences consist of 'Element' objects. There are three kinds of these:
- StringElement contains a single String.
- AndElement contains a list of elements that are concatenated in the given
  order.
- OrElement contains a set of elements that are used alternatively,
  exactly one at a time.
AndElement- and OrElement objects can be nested arbitrarily.
The lowest level (leafs) of any thus created object tree is always formed
by StringElement objects.
OrElements result in variants. Where OrElements meet, the cross product
of their variants is formed.

Here is the class hierarchy:
  abstract class Element
      class StringElement
      class AndElement
      class OrElement
  interface ElementAction
      class Depth
  class ElementMain

ElementMain contains a main program that generates an And/Or sequence
and prints it.

*** DESIGN PATTERN: ***

Element is the abstract superclass of a **Composite**.
StringElement is its leaf type.
AndElement and OrElement are its container (composite) types

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


//————————————— Element ——————————————
/**
```

```
  Interface of the 'Element' classes StringElement, AndElement, OrElement.
  *** DESIGN PATTERN: ***
  Element is the superclass of a Composite pattern.
*/
abstract class Element {
  /** include operation. Adds another element to this element if possible.
  */
  abstract public void add(Element e);

  /** repraesentation operation. Returns a coded representation of the
      Element as a String.
  */
  abstract public String asString();

  /** output operation. Prints the whole Element on System.out
      Each variant is printed on a line of its own.
  */
  public void print() {
     Vector v = variants();
     Enumeration e = v.elements();
     while (e.hasMoreElements())
        System.out.println((String)e.nextElement());
  }

  /** repraesentation operation. Returns a vector in which each element
      contains the String representation of one variant of the
      And/Or sequence.
      Each variant appears exactly once in the vector
  */
  abstract public Vector variants();

  /** Branching operation. Calls that one operation from ElementAction that
      is responsible for (corresponds to) the present subclass.
      *** DESIGN PATTERN: ***
      This is the 'double dispatch' procedure in the visited data
      structure for the Visitor pattern.
  */
  abstract void perform(ElementAction a);
}


//————————— StringElement ——————————
/**
  Element class that contains exactly one String.
  *** DESIGN PATTERN: ***
  StringElement is the (only) leaf class in a Composite pattern.
*/
class StringElement extends Element {
  String contents;                         // Contents of the StringElement

```

```
    /** Generates a StringElement with empty contents.
    */
    public StringElement() {
        contents = new String();
    }

    /** Generates a StringElement with the contents given as argument.
    */
    public StringElement(String s) {
        contents = new String(s);
    }

    /** include operation. Appends e to current String contents
         if e is a Stringelement.
         Throws CastException otherwise.
    */
    public void add(Element e) {
        StringElement s = (StringElement)e;                    // may throw CastException
        contents = contents + s.contents;
    }

    /** repraesentation operation. Returns a coded representation of
         the element as a string of the form
         "contents"
    */
    public String asString() {
        return ("\"" + contents + "\"");
    }

    /** Returns a vector with exactly one element: the String that the
         StringElement contains.
    */
    public Vector variants() {
        Vector v = new Vector();
        v.addElement((Object)contents);
        return v;
    }

    /** Calls the stringAction of the ElementAction given as argument.
         *** DESIGN PATTERN: ***
         perform() is the dispatch operation for Visitor (ElementAction).
    */
    void perform(ElementAction a) {
        a.stringAction(this);
    }
}


//————————— AndElement ——————————
/**
```

```
   And sequence of elements.
   (That means the elements have to be concatenated.)
   *** DESIGN PATTERN: ***
   AndElement is one of the container classes of a Composite pattern.
*/
class AndElement extends Element {
   Vector elems;                    // vector of the elements forming the AndElement

   /** Generates an AndElement without Contents.
   */
   public AndElement() {
      elems = new Vector();
   }

   /** include operation. Adds e as new element to current sequence of
        elements.
   */
   public void add(Element e) {
      elems.addElement(e);
   }

   /** representation operation. Returns a coded representation of the
        Element as a String of the form
        AND(el1&el2&el3)
   */
   public String asString() {
      StringBuffer b = new StringBuffer("AND(");
      Enumeration n = elems.elements();
      while (n.hasMoreElements()) {
         Element e = (Element)n.nextElement();
         b.append(e.asString());
         b.append(n.hasMoreElements() ? "&" : ")");
      }
      return b.toString();
   }

   /** Returns cross product of all variants of all elements in the
        AndElement. The product is formed by concatenating the Strings
        of the variants.
   */
   public Vector variants() {
      Vector result = new Vector();
      Enumeration n = elems.elements();
      while (n.hasMoreElements()) {
         Element e = (Element)n.nextElement();
         Vector 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
            }
            Vector oldresult = result;
            int oldN = oldresult.size();
            int nextN = nextpart.size();
            result = new Vector(oldN * nextN);
            // generate cross product (by String concatenation):
            // result := oldresult x nextpart
            for (int oldI = 0; oldI < oldN; oldI++) {
                String currentold = (String)oldresult.elementAt(oldI);
                for (int nextI = 0; nextI < nextN; nextI++) {
                    String currentnext = (String)nextpart.elementAt(nextI);
                    result.addElement(currentold + currentnext);    // String concat
                }
            }
        }
        return result;
    }

    /** Calls the stringAction of the ElementAction given as argument.
          *** DESIGN PATTERN: ***
          perform() is the dispatch operation for Visitor (ElementAction).
    */
    void perform(ElementAction a) {
        a.andAction(this);
    }
}


//————————— OrElement —————————
/**
    Or set of elements.
    (That means the elements are alternatives.)
    *** DESIGN PATTERN: ***
    OrElement is one of the container classes of a Composite pattern.
*/
class OrElement extends Element {
    Vector elems;                      // vector of the elements forming the OrElement

    /** Generates an OrElement without Contents.
    */
    public OrElement() {
        elems = new Vector();
    }

    /** include operation. Adds e as new element to current set of
          alternative elements.
    */
    public void add(Element e) {
        elems.addElement(e);
```

```
    }

    /** representation operation. Returns a coded representation of the
        Element as a String of the form
        OR(el1|el2|el3)
    */
    public String asString() {
        StringBuffer b = new StringBuffer("OR(");
        Enumeration n = elems.elements();
        while (n.hasMoreElements()) {
            Element e = (Element)n.nextElement();
            b.append(e.asString());
            b.append(n.hasMoreElements() ? "|" : ")");
        }
        return b.toString();
    }

    /** Returns the union of all variants of all elements in the
        OrElement.
    */
    public Vector variants() {
        Vector result = new Vector();
        Enumeration n = elems.elements();
        while (n.hasMoreElements()) {
            Element e = (Element)n.nextElement();
            Vector nextpart = e.variants();
            Enumeration nn = nextpart.elements();
            result.ensureCapacity(result.size() + nextpart.size());
            while (nn.hasMoreElements())
                result.addElement(nn.nextElement());
        }
        return result;
    }

    /** Calls the stringAction of the ElementAction given as argument.
        *** DESIGN PATTERN: ***
        perform() is the dispatch operation for Visitor (ElementAction).
    */
    void perform(ElementAction a) {
        a.orAction(this);
    }
}


//———————— ElementAction ——————————
/**
    Interface for operations on Element structures that must handle the
    parts differently, depending on their type.
    *** DESIGN PATTERN: ***
    Design pattern Visitor: ElementAction is the superclass of all
```

```
  visitors of 'Element' Composite data structures.
*/
interface ElementAction {
  void stringAction(StringElement e);
  void andAction(AndElement e);
  void orAction(OrElement e);
}


//————————- Depth —————————
/**
  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.
*/
class Depth implements ElementAction {
  public int maxDepth = -1;                // Depth of deepest Element
  public int maxAndDepth = -1;             // Depth of deepest AndElement
  public int maxOrDepth = -1;              // Depth of deepest OrElement
  private int depth = 0;                   // current depth (root == 0)

  /** Walks through Element e to initialize itself.
      Components maxDepth, maxAndDepth, and maxOrDepth can be queried
      afterwards.
  */
  public Depth(Element e) {
    e.perform(this);
  }

  public void stringAction(StringElement e) {
    if (maxDepth < depth)
      maxDepth = depth;
  }

  public void andAction(AndElement u) {
    if (maxAndDepth < depth) {
      maxAndDepth = depth;
      if (maxDepth < depth)
        maxDepth = depth;
    }
    iterate(u.elems.elements());
  }


  public void orAction(OrElement o) {
    if (maxOrDepth < depth) {
      maxOrDepth = depth;
      if (maxDepth < depth)
```

```
                maxDepth = depth;
            }
            iterate(o.elems.elements());
        }

        /** Handle all elements contained in an AndElement or OrElement.
        */
        private void iterate(Enumeration n) {
            depth++;                                  // these elements are one level deeper
            while (n.hasMoreElements()) {
                Element e = (Element)n.nextElement();
                e.perform(this);
            }
            depth--;
        }
    }


    //———————— ElementMain ——————————
    /**
        main program. generates an And/Or sequence and prints it.
    */
    public class ElementMain {
        public static void main (String args[]) {
            AndElement u = new AndElement();
            OrElement modal = new OrElement();
            OrElement verb = new OrElement();
            u.add(new StringElement("Who "));
            modal.add(new StringElement("can "));
            modal.add(new StringElement("will "));
            u.add(modal);
            u.add(new StringElement("after all "));
            verb.add(new StringElement("think "));
            verb.add(new StringElement("act "));
            u.add(verb);
            u.add(new StringElement("?"));

            u.print();
            System.out.println(u.asString());
            Depth t = new Depth(u);
            System.out.println(" Depth=" + t.maxDepth +
                               " AndDepth=" + t.maxAndDepth +
                               " OrDepth=" + t.maxOrDepth);
        }
    }
```

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] Douglas Schmidt. Collected papers from the plop '96 and europlop '96 conferences. Technical Report wucs-97-07, Washington University Department of Computer Science, St. Louis, February 1997. (Conference "Pattern languages of programs").

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