UNIVERSITY OF OSLO
Department of Informatics

On the practical use of Software Design Patterns

Dr. Scient. Thesis

Marek Vokáč

November 1, 2004

Abstract

This thesis investigates the *empirical evidence* for the usefulness of design patterns, in the context of software maintenance. Advocates of design patterns claim *general* benefits in the form of higher flexibility, easier maintenance and reduced development time, though there is mostly only anecdotal evidence to support these claims.

The thesis reports on empirical research that has been conducted in three related areas: the effects of design patterns on maintenance *effort* and *correctness*; the connection between design patterns and corrective maintenance *needs*, and the utility of design patterns in organizing the development *process*.

The presence of design patterns in the code can have a significant effect on the effort needed to perform maintenance tasks, and the correctness of the results. The effects are strongly dependent on the expertise of the developers—a pattern that is beneficial to an expert, may cause a sharp increase in effort and a reduced correctness for a novice or intermediate developer. However, some patterns are simple and intuitive enough that they provide benefits to all types of developers, even those who lack knowledge of the particular pattern, or even design patterns in general.

The presence of design patterns can also be correlated with the need for corrective maintenance, i.e., the error rates in the parts of the code conforming to the patterns, relative to the code at large. Some patterns are correlated with complex program structures that are prone to errors, while other patterns result in simple, less connected structures with lower error rates. Assuming that the patterns are used appropriately, this correlation can be used as a predictive or diagnostic tool to determine areas of the code that need particular design attention from expert developers.

Design patterns can also be used in the organization of the development team. Teams are often organized around functional areas or around layers in the code (such as a Web front end or a database-related back end). An alternative, when the architecture of the system is expressed using patterns, is to organize the team around the patterns themselves, so that individual developers work with all the code involved in a pattern, across other boundaries. This can aid in the implementation of complex architectures, for example J2EE applications.

Empirical evidence for these contributions was gathered through a combination of controlled experiments and case studies. Realism was ensured by using paid, professional developers as experimental subjects. In the case studies, industrial projects were examined and a large scale analysis of industrial C++ code performed. Research methods in the field were advanced both by the scale, design and realism of the experiment, and by the creation and validation of a tool for extracting design patterns from C++ code at a high precision and speed.

Acknowledgements

Thanks are due to my thesis supervisors Dag I. K. Sjøberg and Erik Arisholm. Their trust in letting a student manage a large, complex and expensive experiment laid the foundation for the whole thesis. Later discussions on methods and article editing proved invaluable. So did the freedom to pursue the research as I saw fit.

The Simula Research Laboratory is a unique institution, with the opportunities it offers and its emphasis on high quality research. Large experiments with paid professional subjects depend on a flexible administration, a competent technical staff and a funding model found nowhere else. Likewise, studies of large code bases require good hardware.

My fellow students have provided both inspiration, valuable criticism and support when it was most needed. Hopefully I have been able to give something similar in return.

SuperOffice ASA generously provided three years' leave, to complete my PhD, with a part-time position for the duration and a place to come back afterwards. This is not something to be taken for granted, and has provided the security needed to embark on such a large project.

My family has put up with a great deal of overtime work, travel and mental absences. Somehow we even managed to move to a new house in the middle of it all, and passed our 10^{th} anniversary intact.

Finally, a word is due to my late mother, who did not live to see the completion of my degree. Her own PhD was completed in 1949, against a background of interrupted schooling, years of forced labour and a war-torn Europe that did not particularly favour Czech women taking degrees in English. Her mother in turn obtained a PhD in medicine in 1915 in Brno. This family tradition provided much of the inspiration and push to get me started. Claiming that something was too hard for me would have sounded quite hollow in comparison.

Contents

PART I

INTRODUCTION AND SUMMARY

Chapter 1

Introduction

1.1 The Languages of design

The need to correctly design software was apparent from the birth of modern computing. The precursors to the current electronic stored-program computers were mechanical and electromechanical machines, where the basic operations were part of the fundamental design. These operations were then connected by semi-permanent wiring or through plugboards. Additional human interaction was generally needed, though this often took the form of more or less mechanical transfer of stacks of cards from one machine to another. The process is nicely illustrated by Black (2002), who describes the effort needed to set up processing of census data before and during World War II, and by Feynman (1997), who describes a method of parallel computing using punched cards of different colours to distinguish the threads.

The design of these processes was done by highly trained and skilled specialists. Given the visibly high cost of change (including rewiring dozens of machines), it was obviously crucial to arrive at a correct design in advance of the detailed set-up and production.

The advent of the electronic stored-program computer continued this tradition. Computing time was scarce and expensive, and there were severe limits on program size and data storage. Spending significant time on design (on paper) made good economic sense.

With the arrival of a powerful microcomputer on virtually every desk-

top, this tradition was broken, for several reasons.

- Computing power is now essentially free. Every developer "owns" the resources, and compiling or testing a program a dozen times is free in terms of computing cost, if not in terms of time.
- Program development has become a mass activity. Scripting languages like JavaScript and high-level languages such as Visual Basic have seemingly made it possible for people to program without spending years on training.
- Time to market pressure: with an increasing number of companies selling or in other ways being dependent on their software, the pressure to deliver "working" software soon increases. In the tradeoff between easily measurable delivery time and hard to measure quality, quality will often lose.

In parallel with these developments there has been a growing emphasis on the need for proper design, in spite of the pressures to the contrary. Software is becoming immensely complex. Problems of increasing size are being solved, and the underlying technology is growing in complexity almost daily. Many different design methodologies and associated notations have been created and used. Lately there has been a convergence towards UML (Object Management Group, 2004) as a common notation for many design activities. There is less agreement on process, with Agile/XP (Beck, 1999; Beck *et al.*, 2001) and RUP (Jacobson *et al.*, 1999) representing "light" and "heavy" processes respectively.

1.2 Software design as a 'wicked' problem

Rittel and Webber (1973) coined the term "wicked problem" to describe problems (in social planning) that were not tractable by normal, analytical means. Peters and Tripp (1976) argued that software design often represents 'wicked' problems. The characteristics of such problems are worth considering, as they have a great impact on the way we should expect to go about the design of solutions:

1. There is no definitive formulation of a wicked problem. Formulating the problem and the solution are essentially the same thing. Each attempt at creating a solution changes the understanding of the problem.
2. Wicked problems have no stopping rule. Since you cannot define the problem, it is difficult to tell when it is resolved. The problem solving process ends when resources are depleted, stakeholders lose interest or political realities change.
3. Solutions to wicked problems are not true-or-false but good-or-bad. Since there are no unambiguous criteria for deciding if the problem is resolved, getting all stakeholders to agree that a resolution is good enough can be a challenge.
4. There is no immediate and no ultimate test of a solution to a wicked problem. Solutions to wicked problems generate waves of consequences, and it is impossible to know how all of the consequences will eventually play out.
5. Every implemented solution to a wicked problem has consequences. Once the web site is published or the new customer service package goes live, you cannot take back what was on-line or revert to the former customer database.
6. Wicked problems do not have a well-described set of potential solutions. Various stakeholders will have differing views of acceptable solutions. It is a matter of judgment as to when enough potential solutions have emerged and which should be pursued.
7. Every wicked problem is essentially unique. There are no classes of solutions that can be applied to a specific case. "Part of the art of dealing with wicked problems is the art of not knowing too early what type of solution to apply."
8. Every wicked problem can be considered a symptom of another problem. A wicked problem is a set of interlocking issues and constraints that change over time, embedded in a dynamic social context.
9. The causes of a wicked problem can be explained in numerous ways. There are many stakeholders who will have various and

changing ideas about what might be a problem, what might be causing it, and how to resolve it.

10. The planner (designer) has no right to be wrong. A scientist is expected to formulate hypothesis, which may or may not be supportable by evidence. Designers do not have this luxury, they are expected to get things right the first time.

The traditional waterfall model of development (DeGrace and Stahl, 1991) in many ways that assumes the problem to be solved is not wicked. The model consists of a single run through a number of phases, emphasizing the need to make detailed specifications on many levels, followed by an implementation phase. During subsequent testing, conformance to the specification is verified and the system can then be deployed—problem solved.

Since reality seldom is like this, other models have evolved. One cannot escape the fact that any development project would like to proceed from an initial point where the software does not exist, to some kind of "final" point where it exists, works and is deployed. All models therefore contain elements of finding out what to do, doing it and testing it, reminiscent of the waterfall model. What varies is the number, order and repetition of the phases, as well as the criteria for terminating a phase (functionality, quality, time). Thus we have *Spiral* (Boehm, 1986), *Evolutionary* (Gilb, 1985), *Incremental* (Williams, 1975; Whitcomb and Clark, 1989), *Agile* (Beck *et al.*, 2001) and other process models.

A development process is not enough by itself to create software successfully. The design of software is to a large extent concerned with translating abstract requirements into a concrete implementation, an intellectual activity that has so far eluded efforts to automate it on a large scale. Several "paradigms", such as *Structured Programming* (Yourdon, 1976) and *Object-Oriented Analysis and Design* (Booch, 1993) have evolved to describe and define how design should be performed.

Another kind of assistance comes from ways of reusing ideas (as opposed to code) that have been proven in practice. This is the purpose of *Design Patterns*.

1.3 Design patterns

Design Patterns in software engineering seek to capture field-proven solutions to recurring problems. They are semi-formal descriptions that describe a problem, the contexts in which it may appear, and outline a solution and its consequences. A Design Pattern is not a rigorous recipe to be followed to the letter, but it is more than just a loose suggestion. Those patterns that have turned out to have a wide applicability have become classics; others have been relegated to specialist areas or forgotten.

The crucial link between design patterns and wicked problems—including software design—is that patterns are meant to be used *flexibly*, adapting to the particular project, technology and other context present. Well-described patterns also have something to say about the consequences of using them. Thus, they are not absolute prescriptions with rigorous criteria for when and how they should be used; this would not match the reality of most software development, and certainly not the wicked problems that are often encountered.

There are dissenting voices to this interpretation of design patterns. France *et al.* (2004) presented "a rigorous and practical technique for specifying pattern solutions expressed in the unified modeling language (UML)" (quote from Abstract). Here, a need is expressed for *rigorous application* of design patterns to design models. In another recent paper, Rost (2004) argues that a pattern must specify only those parts of a generally applicable solution that are invariant across all known instances of the problem described. Thus, the Factory Method should not be considered a true pattern, since the principle described there (delegation of object creation to some agency that possesses the knowledge of the exact type required) is also present in other patterns, such as Prototype.

However, the present work accepts the starting point that design patterns are a vehicle for practitioners to communicate the principles of proven solutions to recurring problems. It follows that "rigorous application" has little meaning, since the specifics of each instance of a problem are going to be somewhat different from the last one—otherwise,

we should be able to reuse actual code, i.e., a much lower level of abstraction than a pattern.

Similarly, requiring a pattern to be distilled down to the invariant aspects of a solution and those only, over the total possible problem space, would likely reduce its usefulness because of the concurrent loss of context information; it is often by no means obvious that the same solution can in principle be applied to seemingly different problems. Indeed, the patterns Strategy and State in (Gamma *et al.*, 1995, p. 305, 315) give structurally *identical* solutions to different problems. At OOPSLA'98, Agerbo and Cornils (1998), discussed this and also showed that there is a difference in the two patterns caused by their different contexts. Thus we have two valid *applications* of the same fundamental idea, forcefully illustrating this point.

Design Patterns survive by being useful. There are many books describing patterns (an incomplete list includes Gamma *et al.*, 1995; Berry *et al.*, 2002; Fowler, 2002; Alur *et al.*, 2001; Borchers, 2001; Coplien and Schmidt, 1995; Yacoub and Ammar, 2004), and there are a number of conferences dedicated to patterns each year, coordinated by The Hillside Group (2004a): Pattern Languages of Programming (PLoP, USA), ChilliPLoP (USA), EuroPLoP (Europe), KoalaPLoP (Australia), SugarLoafPLoP (Brazil) and MensorePLoP (Japan). In addition, mainstream conferences also have many papers or workshops related to design patterns—OOPSLA (at least 17 related papers in 2004, 4 in 2003) and ECOOP (6 papers in 2004).

The emphasis is on usefulness, and this is highlighted by the way patterns are submitted to the PLoP conferences. Instead of a traditional review process that ends in accept/reject decisions, a "shepherding" process is followed (The Hillside Group, 2004b). In it, the reviewer works with authors of promising pattern papers to refine them into a form good enough for the conference workshop. At the conference, the pattern is dissected and commented by a group of well-prepared peers. This is done using a process known as a "Writer's Workshop" (Schmidt, 2002), which places a heavy emphasis on involving the group and actually relegates the author to be a mute observer. The author can then present the pattern in its final form elsewhere.

Design Patterns can be used to describe many aspects of the software development process. *Structural* and *behavioural* patterns capture the structure and dynamic behaviour of the software itself (Gamma *et al.*, 1995). *Process* patterns relate to the (human) process of creating the software, and describe aspects such as common relations, tensions and conflicts within design teams and how to tackle them (Ambler, 1998). *Cognitive* patterns deal with how the mind works and what approaches one might use to get a better understanding of problems and specifications (Gardner *et al.*, 1998). There are also many patterns specific to the various *domains* for which software is written. Examples include simulation (Ekström, 2000), telecom (Rising and Firesmith, 2001), real-time systems (Douglass, 2002), human-computer interaction design (Borchers, 2001), and even dating (Haugland, 2003).

1.4 Goals, contribution and thesis organization

The primary goal of this research is to advance the understanding of how the usage of design patterns influences software development *in practice*. The choice of empirical—as opposed to theoretical or formal—methods was natural in this context.

The software life cycle begins with an idea or problem that needs solving, and ends with a period of decreasing maintenance leading to obsolescence. The maintenance phase is often the longest of the cycle, and arguably also one of the most expensive, at 40% to 70% of the total cost (see for example Lientz *et al.* (1978); Guimaraes (1983)). This motivated the decision to study the impact of design patterns on this phase, rather than on the initial specification, design and development phases. Only structural and behavioural patterns have been studied, i.e., patterns that specify how the program code should be structured and how the various object behave.

Empirical research on the effects of design patterns in the scientific literature is fairly limited (see Section 3.4, p. 32 in this thesis). Many general claims are made for design patterns, usually with anecdotal evidence to back them up. For almost all patterns, the advantages of flexibility

and power are advocated, typically without much regard to the accompanying complexity that often occurs.

The research in the four papers that constitute this thesis advance the field in the following ways:

1. A controlled experiment (Vokáč *et al.* (2004), p. 50 in this thesis) showed great variability in the ease of learning and use of selected patterns. The subjects were professional consultants, performing maintenance work on four different programs. Of the four patterns tested, one (Observer) was readily used and proved advantageous even to subjects without training. It was even reinvented by one subject during the experiment. Conversely, the Visitor pattern had a significant negative effect on both the maintenance effort and the quality of the results, in spite of specific training given during the experiment. This paper has been published in Empirical Software Engineering (vol. 9, no. 3, 2004).

2. Evaluation of historical data from three years of maintenance of a million-line commercial software product (Vokáč (2005b), p. 139) also showed marked differences in error rates correlated with the presence of design patterns. This gives us the possibility to make predictions about future error rates based the use of patterns, and can be used to better direct the design resources available to a team. This paper has been accepted in IEEE Transactions on Software Engineering.

3. A tool was made that can rapidly and reliably recognize a number of design patterns in C++ code, at a rate of 3×10^6 lines of code per hour. This is at least an order of magnitude faster than other, similar tools and makes it possible to evaluate huge code bases and historical data over long terms. A paper describing this tool (Vokáč (2005a), p. 177) has been accepted for the Journal of Object Technology, to appear in July/August 2005.

4. The usefulness of design patterns as documentation tools to support the reuse of reference application as a basis for a new development project (Vokáč and Jensen (2004), p. 111) was studied in an industrial case. The reference application "PetStore",

published by Sun Microsystems, Inc (2003) is extensively documented with design patterns. The documentation not only aided reuse of the software, but also influenced the organization of the development team. Developers were organized around patterns rather than around layers or functions. This paper was presented at the PROFES 2004 conference and appears in the Proceeedings.

5. The experiment (Vokáč *et al.*, 2004) contributed to the advancement of empirical methods performed by the Software Engineering group at Simula. It illustrated how to tackle the technical and social logistics of getting together 50 participants from 15 different companies for three days, with a complex support infrastructure.

The awareness of a connection between design patterns and error rates, combined with the existence of a tool that can rapidly identify patterns in large programs opens up an avenue for further research. While proprietary source code can be hard to obtain for research use, development companies may be interested in having evaluations performed as part of their internal improvement process, and aggregated results can be published in research journals. The Open Source movement is also a natural provider of raw materials for further research in this field.

If it turns out that there really are consistent correlations between structures and behaviour specified by certain patterns and the frequency of errors, an opportunity exists for building theories that explain these correlations in terms of cognitive and technical factors.

At the same time, such results have an immediate industrial applicability. Already, one project to modify some of the software used in the case study has been completed, and it was in part motivated by the findings of the study. A study is already planned for around the year 2006 to evaluate the long-term effects of these changes, and of other patterns.

Chapter 2

Design Patterns

The term "pattern" has many definitions, several of which are relevant to a study of software design patterns. The following relevant definitions are taken from the Oxford English Dictionary, 2nd Edition:

1. a. 'The original proposed to imitation; the archetype; that which is to be copied; an exemplar' (J.); an example or model deserving imitation; an example or model of a particular excellence.

2. a. Anything fashioned, shaped, or designed to serve as a model from which something is to be made; a model, design, plan, or outline.

6. An example, an instance; esp. a typical, model, or representative instance, a signal example.

7. A precedent, an instance appealed to. *(Obsolete).*

8c. *fig.* An arrangement or order of things or activity in abstract senses; order or form discernible in things, actions, ideas, situations, etc. Freq. with of, as pattern of behaviour = behaviour pattern (see BEHAVIOUR 6), and as second element with defining word.

All of these meanings are reflected in the modern usage of the term Design Pattern in software. However, the term Design Pattern has existed for some time in other contexts; those are the subject of the next section.

2.1 Historical background

The concept of a "Pattern" as a recipe for how certain tasks are to be performed or designed dates back to the seventeenth century. It can best be illustrated by citing a passage from Baer (2002):

> While the emergence of sophisticated accounting and double-entry bookkeeping techniques in the early modern era has been credited with helping to promote economic development, the real estate investment practices of seventeenth-century England, especially London, have not been as closely examined. Evidence suggests that methods of assessing real estate also underwent significant change.
>
> The seventeenth century saw the emergence of "pattern books," which revealed rules of thumb and strategic methods of calculating appropriate yields and value. Although not as technically perfected as double-entry bookkeeping at the time, these books acted as a catalyst for economic development. The origins of pattern books are obscure, but their introduction to the public in the mid-seventeenth century, and their widespread dissemination toward the end of the century, influenced the decisions of a variety of investors who shaped London's remarkable physical growth.

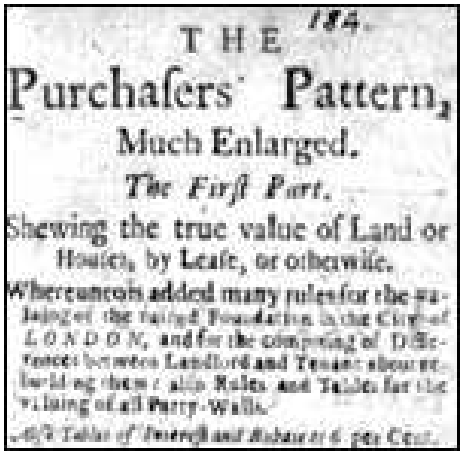
THE 184.
Purchasers Pattern,
Much Enlarged.
The First Part.
Shewing the true value of Land or Houses, by Lease, or otherwise.
Wherunto is added many rules for the ra-
LONDON, and for the compounding of Dif-
ferences between Landlord and Tenant about re-
building them: also Rules and Tables for the
viewing of all Party-Walls.

Figure 2.1: Facsimile of a manuscript from 1677, giving a Pattern for valuation of real estate in London

Already, we see many of the hallmarks of the modern usage of patterns. They specify *rules of thumb* and *strategic* (thus necessarily *abstract*) methods. They are based on *experience*, what has already been found to work. Last but not least, they are read and used by practitioners who thereby gain knowledge and the ability to perform their work better.

The modern usage of Design Patterns is attributed to Christopher Alexander (1977; 1979). He is an architect by profession, and his patterns are architectural. The concept is that there are certain pairs of problems and solutions that appear repeatedly, and that it is possible to capture them in a way that makes this knowledge accessible. Moreover, patters were seen as vehicles to capture the "quality without a name", that which makes some buildings enduringly beautiful and practical.

Figure 2.2: Two patterns from Alexander, one good and one bad.

Figure 2.2 shows two of Alexander's patterns (from A Pattern Language, 1977). The essence of the problem is that people are drawn to the windows of a room, and we need to furnish it accordingly. In "A window place", the sitting group is placed next to the windows; people will drift to the windows naturally, and then find a convenient place to sit there.

"Holes in the wall" can be considered an *anti*pattern—a bad way of doing things. This room has a tension between the easy chairs in one corner, and windows in the opposite corner. The desire to sit and the desire to be close to the window are in tension, and people will not feel quite comfortable in the room.

Alexander's dream was that people, including non-architects, could build better houses by learning a limited number of patterns and applying them wherever they saw fit. The resulting houses and city plans would not be identical, as they would have been from a strict recipe. They would still be individual, while embodying the accumulated wisdom of the specialist architects, like Alexander himself, who would formulate the patterns.

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
>
> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
>
> *Alexander (1977)*

As discussed by Gabriel (1998), Alexander's vision turned out to be very hard to realize, even in circumstances where most of the parameters of building were controlled by Alexander himself (Alexander, 1985). One of the main lessons learned was that a designer or builder does not become an expert simply by having a set of excellent patterns at one's disposal. The same seems to be the case with design patterns in software, and is one of the subjects of this thesis.

2.2 The evolution of software design patterns

The software industry has struggled with the size and complexity of software more or less from the beginning. Despite enormous efforts, large-scale reuse is still problematic. At the same time, it takes both inborn talent and many years of training and experience to create really good software developers.

Alexander's ideas of boiling down specialist experience into a form that was accessible to practitioners (Alexander, 1977, 1979; Alexander *et al.*, 1969), and would guide them without forcing them into a straitjacket appealed to several leading software figures. Ward Cunningham and Kent Beck came up with five patterns for user interfaces as part of

a project in 1987, and published their results at a workshop at OOPSLA'87 (Beck, 1987). At workshops at OOPSLA'91 and '92 (Anderson, 1992), most of the pioneers met and shared ideas.

The Hillside Group first met in 1993 and 1994, with the last meeting dedicated to plan the first PLoP conference, which was held in August 1994. The first edition of "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma *et al.*, 1995) was published in time for OOPSLA'94 and broke Addison-Wesley's previous record for selling technical books at conferences by a factor of 7 (750 copies sold).

In the following years, localized PLoP conferences sprang up in many places—EuroPLoP in Europe, ChiliPLoP in Arizona, MensorePLoP in Japan, KoalaPLoP in Australia, SugarLoafPLoP in Brazil and VikingPLoP in Scandinavia, roughly in order of appearence. Design Patterns have become established tools among software architects, though not so much a the "lower" practitioner levels.

2.2.1 Pattern forms

The presentation forms of patterns have evolved and branched in several directions. The degree of formality differs between forms, and one of the major online references for patterns (Anonymous, 2002) currently lists nine forms as "well known". Since this is a WikiWikiWeb editable by any visitor, and the patterns community is very much alive, this is not a static reference. To show the variety I include a short overview of those forms.

Alexandrian Form
: This form is fairly close to the one used by Alexander (1979). It contains the sections Title, Problem, Discussion, Solution, a Diagram, as well as prologues and epilogues that connect this pattern to other relevant patterns.

Canonical Form
: Canonical form does not necessarily mean the original form, it means the simplest, most basic or primordial form. It is more formal and complete than the Alexandrian form, containing additional sections on Context, Forces, Resulting Context, Ratio-

nale and Known Uses. It is also known as "Coplien Form" as Jim Coplien was one of the more prominent pattern-writers to use it early.

GoF Form

This form was used in the classic book by Gamma *et al.* (1995), and uses different headings from the Canonical Form. However, most of its sections can be mapped to the canonical form, as follows:

GoF	Canonical
Name	Name
Alias	Also Known As
Problem	Intent
Context	Applicability
Forces	Motivation
Solution	Participants, Structure, Collaborations, Implementation
Example	Sample Code
Resulting Context	Consequences
Rationale	
Known Uses	Known Uses
Related Patterns	Related Patterns

Compact Form

While many other forms try to structure and present as much information as possible, the Compact Form goes the other way and targets patterns that can be expressed on a single page. It contains the bare minimum: Context, Problem, Forces, Solution and Resulting Context.

Cockburn PM Form

Alistair Cockburn used this form for his project management patterns. It is more verbal and oriented towards people and processes rather than technical software problems. It has many headings: Title, Thumbnail, Indications, Contraindications, Forces, Do this, Side Effects, Overdose Effect, Related Patterns, Principles, Examples and Reading.

Portland Form

This form is best described by its creators: "Each pattern in the Portland Form makes a statement that goes something like: 'such and so forces create this or that problem, therefore, build a thing-a-ma-jig to deal with them.'

The pattern takes its name from the thing-a-ma-jig, the solution. Each pattern in the Portland Form also places itself and the forces that create it within the context of other forces, both stronger and weaker, and the solutions they require.

A wise designer resolves the stronger forces first, then goes on to address weaker ones. Patterns capture this ordering by citing stronger and weaker patterns in opening and closing paragraphs. The total paragraph structure ends up looking like:

- Having done so and so you now face this problem…
- Here is why the problem exists and what forces must be resolved…

Therefore:

- Make something along the following lines. I'll give you the help I can…
- Now you are ready to move on to one of the following problems…"

Beck Form

Kent Beck made his own variant of the pattern forms, using the headings Title, Context, Problem, Forces, Solution and Resulting Context.

Fowler Form

Martin Fowler used an even more minimal and loose form than the Compact Form in two of his books (Fowler, 2002), keeping only the general outline of Title, Summary and "the bad stuff you avoid by doing this pattern, and how this pattern helps you avoid the bad stuff".

If we look at what is common to all these forms, we recognize the basic structure from Alexander's original. A Design Pattern is a tried and tested solution to a recurring problem, so it should specify what the problem is, outline the solution, and help the reader decide when and how to use it by pointing out relevant contexts, intended effects and side effects. There is no single way that is "right", and different pattern groups have evolved understandings of what they consider to be useful forms.

2.3 Pattern languages

Many patterns are valuable on their own. However, there are many patterns that combine to form a more coherent whole; in this case, we can speak of a pattern *language*. The individual patterns correspond to words, and the language gives rules (grammar) for how they may be combined.

As with the translation of individual patterns into actual code or behaviour, the construction of "sentences" in a pattern language are left as an exercise for the user: the software developer or project manager. Only the practitioner can know which patterns are relevant at any given point, and how they should actually be applied—a very important principle.

An excellent example of a pattern language is described in Scott W. Ambler's book "Process patterns" (Ambler, 1998). It gives a comprehensive set of patterns that can be used in the software development process, with ten distinct, consecutive phases and multiple patterns for each phase. By putting together a set of phases and patterns suitable for each phase, a development team can build and document its own process.

Each of the patterns is complete enough that it has value on its own. However, a well-chosen set of patterns from the language will have a greater value than have the same number of patterns from random, unrelated sources. There may also be higher-level guidance in the language, such as tips on order or decisions points: if you chose a certain

pattern at one point, a whole collection of other patterns may become advantageous or irrelevant.

Chapter 3

Research methods in empirical software engineering

This chapter summarizes the research methods of, and constraints on empirical software engineering as a discipline.

3.1 Science versus industrial practice

Unlike fields like physics or mathematics, research on software engineering studies the production of *intellectual* artefacts by humans. The "reality" being studied at any point is the combination of current industrial practice and theoretical work in both industry and academia. It is constantly changing and is to a certain extent dependent on local cultures and habits.

There are several differences between software artefacts and other human-built structures such as bridges or aircraft, but the main one is that the software is primarily an intellectual product, relatively independent of its actual physical storage or execution units. At the same time, software, unlike abstract mathematics, is expected to interact with the physical world in a manner yielding some predefined result. Since software is primarily created for commercial or practical reasons, its development is directed towards a concrete goal, constrained by time and resources in a way that abstract mathematical research is not. This is one of the criteria of "wicked problems", listed in 1.2.

An aircraft is a structure whose construction is constrained by physical laws such as gravitation, aerodynamics and the strength of materials. It is also highly visible, and visibly complex. The serious consequences of major flaws in the design are obvious and motivate a corresponding attention to design and construction. Development of a new aircraft, even one that is largely derived from an existing model, costs billions of dollars.

Apart from its development effort, a software program is only constrained by the size and speed of the computers it runs on, and the exponential development in these parameters is well-known and without equivalence in the "physical" world. Software has therefore proceeded from a tractable level of complexity (1950's) to a point where a desktop operating system contains 10^7 lines of code, a size obviously beyond being fully testable or predictable as a system. At the same time, software is largely invisible, and its complexity remains hidden. Consequently, the need for careful design suffers from the same lack of visibility, and the money available for development is one or more orders of magnitude lower than for correspondingly complex mechanical devices.

Brooks (1987) highlighted all of these differences in his paper "No silver bullet", and they have only become greater with the exponential increase in computing power and complexity since then (roughly a factor 250 for typical CPU power, 500 for memory and 10 000 for disk space).

In the software industry, the choice of development methods and processes is influenced by many factors. The personal preferences and qualifications of the developers have a major impact on the process that is actually followed, a process that is sometimes quite different from the one specified by the organization. Other factors include perceived benefits (that may be motivated more by politics than technology) from adhering to a certain process, requirements by customers, and experience from other projects.

Software is a term that has a wide meaning. Viewed narrowly it can simply represent the code that is actually running in a computer, though usually in its high-level form and not compiled machine code.

However, it is also often used in a wider sense, encompassing formal and semi-formal models, designs and other materials directly related to the program code.

All of the above combine into a fairly chaotic reality, and makes software engineering research more reminiscent of social studies than physics. On the other hand, current computers are fundamentally deterministic machines, and the languages we use to program them have fairly simple grammars, expressible in formal systems. Similarly, more abstract design and specification languages exist (such as SDL) that can be automatically translated to executable code, with designs that can be mechanically verified to have certain properties. This corresponds better to the world-view of the mathematical community, and scientific methods based on rigorous derivation of theories from first principles.

There is, therefore a certain tension between the scientific and industrial approaches to software engineering—between the wish to apply methods and reasoning models from mathematics and the physical sciences, and the presence of intractable and hidden complexity plus a large number of human and social factors that are hard to characterize, control or repeat.

Good opportunities for research do exist where there are enduring needs, such as the need to go from informally specified requirements via general solutions to specific, programmable designs. This transition is one of the fundamental and difficult problems of software engineering. It is also quite unavoidable. Design patterns represent one way of attacking this problem, and are of interest to both industrial practitioners and researchers.

3.2 Empirical research on software engineering

The empirical method is one of several different research models that can be applied to software engineering or other fields. Quoting from Adrion (1992),

> Part of the problem with SE research methodology lies in how

> one defines the boundaries of the field. While typical engineering research builds on principles from clearly defined scientific disciplines—physics, chemistry, etc.—the boundary between SE and its scientific bases in programming languages, data structures, algorithms, operating systems, etc. is much less clearly defined. Many research "achievements" in SE could actually be said to be advances in the underlying science base, rather than in the engineering of software.
>
> Thus, methodological problems arise, since one is, at times, attempting to advance both the science underlying SE and the engineering practice simultaneously. In addition, unlike many other engineering disciplines, there is no clear boundary between engineering and management issues in SE. To develop management principles and refine engineering practice together can also lead to methodological conflicts.

Four different models are cited by Adrion as applicable to software engineering research:

1. The scientific method: observe the world; propose a model or theory of behaviour, measure and analyze, validate hypotheses of the model or theory; and if possible, repeat.
2. The engineering method: observe existing solutions, propose better solutions, build or develop, measure and analyze, repeat until no further improvements are possible.
3. The empirical method: propose a model, develop statistical or other methods, apply to case studies, measure and analyze, validate the model, repeat.
4. The analytical method: propose a formal theory or set of axioms, develop a theory, derive results and if possible compare with empirical observations.

These models can be characterized along two axes: a) The presence or absence of a theoretical model; and b) whether observation of the world takes place before or after the theoretical model is proposed. In practice, since software engineering is an activity that we are demonstrably unable to characterize and deduce from *a priori* formal models, a mix of the scientific, empirical and possibly engineering methods is a nat-

ural (maybe inevitable) choice. For sharply defined, limited areas the analytical model may still be applicable, but not to software engineering in general.

3.3 Empirical research methods

The three most commonly used study methods in empirical research are the Survey, the Case Study and the Experiment.

A *survey* is research in the large: it covers broad tendencies over a large number of projects, methods or subjects. Depending on the openness of the questions, it can be used as a theory-testing or theory-generating study. However, demonstrating causality is difficult or impossible using surveys, and the validity of a survey is threatened both by sample selection and question formulation.

Achieving truly random samples is usually very hard, as one cannot generally force subjects (chosen through randomization) to respond. Unless the response rate is carefully monitored and any systematic non-responsiveness compensated for, the result will be skewed samples (for instance, those subjects who were interested enough to respond!). This can be ameliorated, for instance, by offering a reward for participation. However, this raises other problems, such as whether respondents then feel compelled to give answers that (in their opinion) give the researchers "value for money". Due to the lack of close contact, it is not possible to know in detail how the subjects interpret the questions, and therefore what they are actually answering.

In a *case study*, the researcher wants to understand a particular situation or process in depth. The target project or organization is chosen through a deliberate process, either because it is considered to be typical of a field, or because it illustrates one particular point (Yin, 2003). The choice is affected by the kind of generalization that is intended. In order to be able to claim external validity for a case study, it must be possible to argue strongly that it is in all significant ways typical of the intended target population (which must also be carefully specified). On the other hand, a case study performed on an extreme project can be

used to illustrate a border or end point to the validity of an underlying theory. The researcher may observe concurrently with the project, retrospectively, or even be an active participant (Action Research). Case studies generally try to disturb the target project as little as possible, but some impact from data collection, interviews or measurements is usually unavoidable.

Controlled experiments are particularly useful as theory-testing devices. This implies that before an experiment can be designed and performed, sufficient knowledge must already have been gathered to formulate a theory. From the theory, hypotheses are derived, and then subjected to test through the experiment (Christensen, 2001). Experiments are used in all scientific disciplines, and fairly detailed guidelines for their use in software engineering research exist (Kitchenham *et al.*, 2002). Since they are also one of the main ingredients of the present research (and, indeed, of the research strategy of the Software Engineering group at Simula Research Laboratory), they are considered in more detail below.

3.3.1 Experiments in software engineering research

Formal experiments constitute a rigorous approach to testing hypotheses and the theories from which they are derived. However, the validity of an experiment still faces numerous threats (Sjøberg *et al.*, 2002, 2003; Kitchenham *et al.*, 2002). The most usual threats in software engineering research are:

1. *Population characterization.* In order to generalize a result, the population to which it applies (and from which we are selecting our sample of subjects) must be clearly defined. This is difficult—is the population the sum of all developers? Developers using a particular language? Developers with a certain educational background, working on a particular type of software?

   Since software engineering is an activity that is intensely dependent on human factors, the number of potentially confounding variables is large and the definition of a population correspondingly difficult. Ultimately, the theory that we are trying to test

through the experiment should provide a framework for characterizing an "interesting" population. Otherwise, a large degree of arbitrariness and therefore uncertainty will remain.

2. *Sample selection.* Statistical theory states that inferences can only be made about a population if the study subjects constitute a random sample. Again, this is difficult to achieve.

   Students have often been used as subjects, for pragmatic reasons. They are inexpensive or free, and they generally do as they are told, especially if the researcher is also their tutor and the one who sets their grades. However, recent research (Arisholm and Sjøberg, 2004) shows that students can give significantly different results from professionals, under the same conditions, and that the differences may not be as expected (i.e., the students may actually do better than the professionals, where one would usually assume the opposite). Significant ethical concerns can also arise in such situations.

   When using professionals as subjects, it is necessary to provide payment on an industrial scale. Otherwise, the developers (if self-employed) or their employers will incur a monetary loss by participating, and the chance of having a representative sample is very low. This makes experiments with many subjects quite expensive, relative to the traditional funding of software engineering research. Still, the amounts of money involved are trivial compared to the expense of any kind of industrial pilot plant, clinical study in pharmacology or physics experiment, let alone the development of a new car engine—arguably all fields with impacts of the same order of magnitude as software development.

3. *Materials and tasks*. In his famous paper "No silver bullet", Brooks (1987) argued that the *essence* of software lies in its complexity, conformity, changeability and invisibility. If we accept this, experiments must reproduce these significant aspects in order to be valid.

   Reproducing complexity is particularly difficult, since the use of complex and large experimental materials drastically prolongs

the time it takes to prepare and execute the experiment. Long execution times combined with large numbers of paid subjects are outside the scope of most research budgets. However, there is also little or no tradition of *applying* for grants to do such experiments. Other research fields have been successful in establishing the need for, and results from, rigorous experiments and there is no fundamental reason why software engineering should not do the same.

The task of properly preparing materials, such as code to be maintained, is very difficult if the code is to be large and complex, and still have sharply defined qualities in the areas we wish to measure.

Usually, experiments last on the order of hours. In this research (Vokáč *et al.*, 2004), that was increased to three days. Later experiments at Simula have successfully extended the duration to several months.

The difficulties of designing and conducting controlled experiments combined with their expense is probably the reason why there are relatively few of them. An extensive survey of the main software engineering journals and conferences (Sjøberg *et al.*, 2004) found 113 papers on experiments out of a total of 5433 papers over the last ten years, in nine journals and four conferences. The vast majority of these concerned software life-cycle/engineering (48%), methods (32%) and Project/product management (8%). Only 18% of the experiments (N=21) used only professionals as subjects. Only 2.2% of the total subjects were paid; more significantly, in 65% of all surveyed experiments the rewards, if any, were not reported and we can therefore form no opinion of whether the reward may have influenced the results. About 70% of the experiments were conducted on constructed (i.e, artificial) materials, and about 65% of the experiments concerned inspections or other tasks that do not modify the code.

A single experiment, however well designed and executed, does not constitute an absolute result, universally applicable. Thus, replication of experiments is considered essential in most disciplines. For instance, the repeated surveys of smoking have together provided incontrovert-

ible evidence of the harmful effects, where single studies could be disputed. However, replicated experiments in software engineering are rare (20 out of 103 in Sjøberg *et al.* (2004)). Lindsay and Ehrenberg (1993) advance some reasons why replication is seldom undertaken, and present a theory of replication. Their field is the social sciences, but the arguments are applicable to software engineering research as well.

Replication is often considered to have a lower prestige than original experiments. The emphasis on originality can be seen everywhere, including the definition of what constitutes good PhD research (!). This may partly be due to the misunderstanding that *replication* is the same as simple *repetition*.

An identical replication of an experiment is neither desirable nor possible in software engineering research[1]. One cannot make the same people perform the exact same task twice under the exact same conditions, with no leakage of experience, knowledge or weariness across the experiments.

However, it is critical to understand how a replication differs from its predecessors. Well-chosen differences contribute to the knowledge in the field; ill-chosen or uncontrolled differences may invalidate the replication and in effect turn it into a new experiment with results that cannot be compared with the original experiment.

The experiment reported in Vokáč *et al.* (2004) is a replication, and illustrates how replication is different from mere repetition. We wished to increase the realism of the experiment (Sjøberg *et al.*, 2002), and did so in two ways: 1) we used a real programming environment, instead of annotations to paper printouts, and 2) our subjects came from multiple consultancy companies and were paid for their participation, instead of being volunteers from a single company. A more sophisticated statistical method was used to analyze the raw data.

However, we used the same general design of the experiment and the exact same set of four tasks. The program code was identical to that

1. Identical replication may be possible in physics or chemistry, but then it is used to characterize the precision of the measurements or experimental set-up, or to discover variability in the underlying phenomenon, or missed factors.

of the original experiment, except for corrections of minor errors. The same course on Design Patterns was taught by the same person (Walter Tichy), using the same course materials.

Using the terminology of Lindsay and Ehrenberg (1993), this experiment was a "close" replication with the differences optimized for increasing the realism of the results. Still, some other differences were inevitable: German and Norwegian developers come from different cultures, and the underlying programming languages and facilities have evolved during the four years that separated the original and the replication.

Some of the observed discrepancies in the results are probably caused by these differences. On the other hand, since the raw data from the original experiment were available in full, the new analysis approach could be applied to them, to eliminate discrepancies caused by differing statistical methods. It is thus both possible and useful to perform replications.

At the Simula research Laboratory, the Software Engineering group has a long-running series of experiments on the effects of delegated versus centralized control styles on maintainability (Arisholm and Sjøberg, 2004). Close to 200 subjects have participated in multiple replications, the latest addition being the use of pair programming. This is a unique set of materials that will together have much greater impact than any one of the experimental runs by itself.

3.4 Empirical research on design patterns

Design patterns have been in use in software since around 1994, and they have been widely publicized through books, seminars, conferences and in educational curricula. However, the amount of solid empirical research around design patterns is quite limited.

The central figures of the patterns community have in general tended to avoid hyperbole; however, some fairly wide claims are still made:

> OK, let's get it out up front. Patterns are not, repeat not, a silver

> bullet. They certainly won't single-handedly solve the software crisis! Patterns will not create expert designers out of novices and suddenly, magically, make everyone in your organization a guru!
>
> *Linda Rising*, Rising (1998)

> It should help novices to act as if they were—or almost as if they were—experts on modest-sized projects, without having to gain many years of experience.
>
> *Frank Buschmann*, Buschmann *et al.* (1996)

> Certain tried-and-true solutions to design problems can be (and have been) expressed as a set of principles, heuristics or patterns—named problem-solution formulas that codify exemplary design principles. This book, by teaching patterns, supports quick learning and skillful use of these fundamental object-oriented design idioms.
>
> *Craig Larman*, Larman (2001)

Most of the documentation of design patterns is in the form of books or articles, written for the general developer audience. Backing for claims is given in anecdotal form, with stories from the author's own experience.

The focus of the present research is on experimental investigation of how use of design patterns affects the maintainability of software. Increased maintainability, through both lower error rates and increased flexibility and adaptability to new circumstances, is one of the perceived advantages of design patterns.

Academic research on this aspect is limited. Prechelt and Unger (1999) set out a program of research into patterns, and performed several experiments (Prechelt *et al.*, 2001, 2002). Bieman *et al.* (2001, 2003) have performed several industrial case studies. Further case studies have been conducted by several groups (Schmidt and Stephenson, 1995; Neumann and Zdun, 2002; Chu *et al.*, 2000). The subjects of the studies are industrial systems of up to 30 000 LOC, but they address re-engineering or construction concerns, not maintenance over extended periods.

The conclusions from these studies are that pattern use *per se* cannot

be said to be exclusively beneficial, nor will even the qualified use of patterns guarantee below-average error rates or maintenance effort. Of course, it is hard to know what would have happened if patterns had been used differently or not at all in the studied cases; the replicated experiment in this thesis (Vokáč *et al.*, 2004) tested patterns against "equivalent" designs to illuminate this.

Other research on design patterns has focused on tools to support the use of patterns in the development process—either during design/coding, or reverse engineering. Reverse engineering tools have been created by several groups (Kramer and Prechelt, 1996; Florijn *et al.*, 1997; Bansiya, 1998; Antoniol *et al.*, 1998, 2001; Keller *et al.*, 1999; Schauer and Keller, 1998; Albin-Amiot *et al.*, 2001; Guéhéneuc and Albin-Amiot, 2001; Balanyi and Ferenc, 2003). Most existing tools have been tested on code sizes up to 10 000 LOC, and usually less (or not specified). Running times are usually not specified, so that it is hard to extrapolate.

Commercial software often runs into millions of lines of code, and it is arguably when the code is largest that the need for reverse engineering tools is greatest. This was one of the motivations for the research reported in Vokáč (2005a,b), where large-scale reverse engineering was needed to obtain the raw data for an evaluation of the effects of design pattern usage on error rates.

3.5 Summary of research papers and methods

The four papers in thesis cover several different methods.

In order to obtain detailed information on the effects of certain selected patterns, an experiment was performed (Vokáč *et al.* (2004), p. 50). To increase the validity and value of the experiment, it was a replication of an earlier experiment. Several changes were made relative to the original, the most important being the use of paid professionals, and a real, well instrumented programming environment.

The possible effects of using design patterns on the development pro-

cess itself were investigated through a case study (Vokáč and Jensen (2004), p. 111); using a traditional experiment here would have required very large resources in order to run the project several times "with" and "without" design patterns being involved.

A second case study was conducted to investigate the effects of design patterns on a large, commercial software product (Vokáč (2005b), p. 139). The knowledge gained here partly confirmed that from the experiment, but some differences were uncovered as well—mainly the increased error rate associated with the Observer pattern, which contradicts the results from the experiment. However, one of the defining differences between the experiment and the case study as research methods is that the case study allows much larger artefacts to be investigated—and the problems with Observer were related to the size and complexity of the classes involved. The benefits of using more than one approach to research are well illustrated by this juxtaposition.

Finally, in order to perform the second case study efficiently (or indeed at all), a method to extract design patterns from C++ code was needed. This led to the development of a new tool, which was validated as part of the study (Vokáč (2005a), p. 177).

To conclude, using several different and complementary research methods, the research reported here advances both the understanding of the effects of design patterns, proposes that the usage of patterns in code can be used to predict error rates, and advances the state of the art of both pattern-recovering tools and software engineering experiments.

Bibliography for Introduction and summary

Adrion, W. R., 1992. **Research Methodology in Software Engineering**. In: Tichy, W. F., Habermann, N., Prechelt, L. (Eds.), **Dagstuhl Workshop on Future Directions in Software Engineering**. ACM SIGSOFT, Schloss Dagstuhl, pp. 36–37.

Agerbo, E., Cornils, A., 1998. **How to Preserve the Benefits of Design Patterns**. In: **OOPSLA '98: Conference on Object Oriented Programming Systems Languages and Applications**. Vol. 33 of SIGPLAN Notices. ACM Press, Vancouver, British Columbia, Canada, pp. 134–143.

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y. G., Jussien, N., 2001. **Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together**. In: **ASE 2001: 16th Annual International Conference on Automated Software Engineering**. IEEE CS Press, San Diego, CA, USA, pp. 26–29.

Alexander, C., 1977. **A Pattern Language: Towns, Buildings, Construction**. Center for Environmental Structure. Oxford University Press, New York, ISBN: 0195019199.

Alexander, C., 1979. **The Timeless Way of Building**. Center for Environmental Structure. Oxford University Press, New York, ISBN: 0195024028.

Alexander, C., 1985. **The Production of Houses**. Oxford University Press, New York, ISBN: 0195032233.

Alexander, C., Hirshen, S., Ishikawa, S., Coffin, C., Angel, S., 1969. **Houses Generated by Patterns**. Center for Environmental Studies, Berkely.

Alur, D., Crupi, J., Malks, D., 2001. **Core J2EE Patterns**. Prentice-Hall, Upper Saddle River, NJ, USA, ISBN: 0130648841.

Ambler, S. W., 1998. **Process Patterns**. The Press Syndicate of the University of Cambridge, Cambridge, United Kingdom, ISBN: 0521645689.

Anderson, B., 1992. **Towards An Architecture Handbook**. In: **OOPSLA '92: Conference on Object Oriented Programming Systems Languages and Applications**. Vol. 27 of SIGPLAN Notices, Issue 10. ACM Press, Vancouver, British Columbia, Canada, pp. 109–113.

Anonymous, 2002. **Pattern Forms**. URL `http://c2.com/cgi/wiki?PatternForms`

Antoniol, G., Casazza, G., Di Penta, M., Fiutem, R., 2001. **Object-Oriented Design Patterns Recovery**. Journal of Systems and Software 59 (2), 181–196.

Antoniol, G., Fiutem, R., Cristoforetti, L., 1998. **Using Metrics to Identify**

Design Patterns in Object-Oriented Software. In: **Metrics 1998: Fifth International Software Metrics Symposium, 1998.** IEEE Computer Society, Bethesda, Maryland, USA, pp. 23–34.

Arisholm, E., Sjøberg, D., 2004. **Evaluating the Effect of a Delegated Versus Centralized Control Style on the Maintainability of Object-Oriented Software**. IEEE Transactions on Software Engineering 30 (8), 521–534.

Baer, W. C., 2002. **The Institution of Residential Investment in Seventeenth-Century London**. Business History Review 76 (Autumn 2002), 515–552.

Balanyi, Z., Ferenc, R., 2003. **Mining Design Patterns from C++ Source Code**. In: **ICSM'03: International Conference on Software Maintenance**. IEEE Computer Society, Amsterdam, The Netherlands, pp. 305–315.

Bansiya, J., June 1998 1998. **Automating Design-Pattern Identification**. Dr. Dobb's Journal 23 (6), 20–2, 24, 26, 28.

Beck, K., 1987. **Using a Pattern Language for Programming**. In: Kerth, N. L., Hogg, J., Stein, L., Porter, H. H. (Eds.), **OOPSLA'87: Addendum to the Proceedings**. ACM Press, Orlando, Florida, USA, p. 16.

Beck, K., 1999. **Extreme Programming Explained: Embrace Change**. Addison Wesley, Boston, MA, USA, ISBN: 0201616416.

Beck, K., Beedle, M., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D., 2001. **Agile Manifesto**. URL `http://www.agilemanifesto.org/`

Berry, C., Carnell, J., Juric, M., Kunnumpurath, M., Nashi, N., Romanosky, S., 2002. **J2EE Design Patterns Applied**. Wrox Press Ltd, Hoboken, NJ, USA, ISBN: 1861005288.

Bieman, J., Jain, D., Yang, H., 2001. **OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study**. In: **ICSM 2001: IEEE International Conference on Software Maintenance, 2001**. IEEE Computer Society, Firenze, Italy, pp. 580–589.

Bieman, J., Straw, G., Wang, H., Munger, P., Alexander, R., 2003. **Design Patterns and Change Proneness: An Examination of Five Evolving Systems**. In: **METRICS '03: Ninth International Software Metrics Symposium, 2003.** IEEE Computer Society, Sydney, Australia, pp. 40–49.

Black, E., 2002. **IBM and the Holocaust**. Time Warner Paperback, ISBN: 0751531995.

Boehm, B., 1986. **A Spiral Model of Software Development and Enhancement**. ACM SIGSOFT Software Engineering Notes 11 (4), 14–24.

Booch, G., 1993. **Object-Oriented Analysis and Design**, 2nd Edition. Pearson Education, Upper Saddle River, NJ, USA, ISBN: 0805353402.

Borchers, J., 2001. **A Pattern Approach to Interaction Design**. John Wiley & Sons, Hoboken, NJ, USA, ISBN: 0471498289.

Brooks, F. P. J., 1987. **No Silver Bullet: Essence and Accidents of Software Engineering**. IEEE Computer 20 (4), 10–19.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. **Pattern-Oriented Software Architecture**. Wiley, Chichester, ISBN: 0 471 95869 7.

Christensen, L. B., 2001. **Experimental Methodology**, 8th Edition. Allyn & Bacon, Boston, MA, USA, ISBN: 0-205-30832-5.

Chu, W. C., Lu, C. W., Shiu, C. P., He, X. D., 2000. **Pattern-Based Software Reengineering: A Case Study**. Journal of Software Maintenance—Research and Practice 12 (2), 121–141.

Coplien, J., Schmidt, D., 1995. **Pattern Languages of Program Design**. Addison Wesley, Boston, MA, USA, ISBN: 0201607344.

DeGrace, P., Stahl, L. H., 1991. **Wicked Problems, Righteous Solutions**. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, ISBN: 013590126X.

Douglass, B. P., 2002. **Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems**. Addison-Wesley, Boston, MA, USA, ISBN: 0201699567.

Ekström, U., 2000. **Design Patterns for Simulations in Erlang/OTP**. Master's thesis, Uppsala University, Sweden.

Feynman, R., 1997. **Surely You're Joking, Mr Fenyman**. W. W. Norton & Company, New York, USA, ISBN: 0393316041.

Florijn, G., Meijers, M., van Winsen, P., 1997. **Tool Support for Object-Oriented Patterns**. In: **ECOOP '97: European Conference on Object-Oriented Programming**. Vol. 1241 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Heidelberg, pp. 472–495.

Fowler, M., 2002. **Patterns of Enterprise Application Architecture**. Addison Wesley Professional, Boston, MA, USA, ISBN: 0321127420.

France, R., Kim, D.-K., Ghosh, S., Song, E., 2004. **A UML-Based Pattern Specification Technique**. IEEE Transactions on Software Engineering 30 (3), 193–206.

Gabriel, R., 1998. **The Failure of Pattern Languages**. In: Rising, L. (Ed.), **The Patterns Handbook**. Cambridge University Press, Melbourne, Australia,

pp. 333–343, ISBN: 0-521-64818-1.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Boston, MA, USA, ISBN: 0201633612.

Gardner, K. M., Rush, A., Crist, M. K., Konitzer, R., Teegarden, B., 1998. **Cognitive Patterns**. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0-521-64998-6.

Gilb, T., 1985. **Evolutionary Delivery Versus the "Waterfall Model"**. ACM SIGSOFT Software Engineering Notes 10 (3), 49–61.

Guéhéneuc, Y.-G., Albin-Amiot, H., 2001. **Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects**. In: **TOOLS 39: 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001**. Santa Barbara, CA, USA, pp. 296–305.

Guimaraes, T., 1983. **Managing Application Program Maintenance Expenditures**. Communications of the ACM 26 (10), 739–746.

Haugland, S., 2003. **Dating Design Patterns**. Published by Solveig Haugland, ISBN: 0974312002.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. **The Unified Software Development Process**. Addison-Wesley Professional, Boston, MA, USA, ISBN: 0201571692.

Keller, R., Schauer, R., Robitaille, S., Pagé, P., 1999. **Pattern-Based Reverse-Engineering of Design Components**. In: **ICSE '99: 1999 International Conference on Software Engineering**. ACM Press, Los Angeles, CA, USA, pp. 226–235.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El-Emam, K., Rosenberg, J., 2002. **Preliminary Guidelines for Empirical Research in Software Engineering**. IEEE Transactions on Software Engineering 28 (8), 721–734.

Kramer, C., Prechelt, L., 1996. **Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software**. In: **Third Working Conference on Reverse Engineering, 1996**. IEEE Computer Society, Monterey, CA, USA, pp. 208–215.

Larman, C., 2001. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process**, 2nd Edition. Prentice Hall, Upper Saddle river, NJ, USA, ISBN: 0130925691.

Lientz, B. P., Swanson, E. B., Tompkins, G. E., 1978. **Characteristics of Applica-**

tion Software Maintenance. Communications of the ACM 21 (6), 466–471.

Lindsay, R., Ehrenberg, A., 1993. **The Design of Replicated Studies**. The American Statistician 47 (3), 217–228.

Neumann, G., Zdun, U., 2002. **Pattern-Based Design and Implementation of An XML and RDF Parser and Interpreter: A Case Study**. In: **ECOOP '02: 16th European Conference on Object-Oriented Programming**. Vol. 2374 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, University of Mlaga, Spain, pp. 392–414.

Object Management Group, 2004. **UML 2.0 Specifications**. URL `http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML`

Peters, L. J., Tripp, L. L., 1976. **Is Software Design Wicked?** Datamation 22 (5), 127–.

Prechelt, L., Unger, B., 1999. **Methodik und Ergebnisse einer Experimentreihe über Entwurfsmuster**. Informatik - Forschung und Entwicklung 14 (2), 74–82.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., Votta., L. G., 2001. **A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions.** IEEE Transactions on Software Engineering 27 (12), 1134–1144.

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W. F., 2002. **Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance**. IEEE Transactions on Software Engineering 28 (6), 595–606.

Rising, L., 1998. **The Patterns Handbook**. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0521648181.

Rising, L., Firesmith, D. G., 2001. **Design Patterns in Telecommunications Software**. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0521790409.

Rittel, H. W. J., Webber, M. M., 1973. **Dilemmas in a General Theory of Planning**. Policy Sciences 4 (2), 155–169.

Rost, J., 2004. **Is "Factory Method" Really a Pattern?** ACM SIGSOFT Software Engineering Notes 29 (5), 1–1.

Schauer, R., Keller, R., 1998. **Pattern Visualization for Software Comprehension**. In: **IWPC '98: 6th International Workshop on Program Comprehension, 1998**. pp. 4–12.

Schmidt, D., 2002. **How to Hold a Writer's Workshop**. URL `http://www.cs.wustl.edu/ schmidt/writersworkshop.html`

Schmidt, D., Stephenson, P., 1995. **Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms**. In: **ECOOP '95: European Conference on Object-Oriented Programming**. Vol. 952 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Århus, Denmark, pp. 399–423.

Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E., Vokáč, M., 2002. **Conducting Realistic Experiments in Software Engineering**. In: **ISESE 2002: First International Symposium on Empirical Software Engineering**. IEEE Computer Society, Nara, Japan, pp. 17–26.

Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Vokáč, M., 2003. **Challenges and Recommendations When Increasing the Realism of Controlled Software Engineering Experiments**. In: Conradi, R., Wang, A. I. (Eds.), **ESERNET 2001-2002**. Vol. 2765 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, pp. 24–38.

Sjøberg, D. I. K., Kampenes, V. B., Hannay, J. E., Hansen, O., Karahasanovic, A., Liborg, N.-K., Rekdal, A. C., 2004. **A Survey of Controlled Experiments in Software Engineering**. Submitted to IEEE Transactions on Software Engineering.

Sun Microsystems, Inc, 2003. **Java Pet Store Demo 1.1.2**. URL `http://java.sun.com/blueprints/code/jps11/docs/index.html`

The Hillside Group, 2004a. **Design Patterns Conferences**. URL `http://hillside.net/conferences/`

The Hillside Group, 2004b. **Shepherding**. URL `http://hillside.net/shepherding.html`

Vokáč, M., 2005a. **A Tool for Recovering Design Patterns from C++ Code, and its Application in a Case Study**. Journal of Object Technology, To appear July/August 2005.

Vokáč, M., 2005b. **Defect Frequency and Design Patterns: An Empirical Study of Industrial Code**. IEEE Transactions on Software Engineering Accepted for publication.

Vokáč, M., Jensen, O., 2004. **Using a Reference Application with Design Patterns to Produce Industrial Software**. In: Bomarius, F., Iida, H. (Eds.), **Product Focused Software Process Improvement**. Vol. 3009 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Kansai Science City, Japan, pp. 333–347.

Vokáč, M., Tichy, W., Sjøberg, D. I. K., Arisholm, E., Aldrin, M., 2004. **A Controlled Experiment Comparing the Maintainability of Programs De-**

signed with and Without Design Patterns: A Replication in a Real Programming Environment. Empirical Software Engineering 9 (3), 149–195.

Whitcomb, M., Clark, B., 1989. **Pragmatic Definition of An Object-Oriented Development Process for Ada**. In: **Tri-Ada '89: Ada Technology in Context: Application, Development, and Deployment**. ACM Press, Pittsburgh, Pennsylvania, United States, pp. 380–399.

Williams, R. D., 1975. **Managing the Development of Reliable Software**. In: **International Conference on Reliable Software**. ACM Press, Los Angeles, California, pp. 3–8.

Yacoub, S. M., Ammar, H. H., 2004. **Pattern Oriented Analysis and Design (POAD): Composing Patterns to Design Software Systems**. Addison Wesley, Boston, MA, USA, ISBN: 0201776405.

Yin, R., 2003. **Case Study Research, Design and Methods**, 3rd Edition. Sage Publications, Thousand Oaks, CA, USA, ISBN: 0-7619-2552-X.

Yourdon, E., 1976. **How to Manage Structured Programming**. Prentice Hall PTR, Indianapolis, Indiana, USA, ISBN: 0917072022.

PART II

PEER-REVIEWED PAPERS

Paper 1

A controlled experiment comparing the maintainability of programs designed with and without design Patterns—a replication in a real programming environment

This paper appeared in Empirical Software Engineering, vol. 9, issue 3, pp. 149–195, 2004. [1]

1. The text of all papers has been reformatted to fit in this thesis. The varying styles of bibliographical references practiced by the different journals have forced some sentences to be rewritten to make sense with the citation style used in the thesis. Sections have also been renumbered to fit into the overall table of contents. The format changes have also caused some tables and figures to move slightly. Otherwise, the text is identical to the published version.
All papers have either been published or accepted; there are no outstanding revisions.

A controlled experiment comparing the maintainability of programs designed with and without Design Patterns—a replication in a real programming environment

Marek Vokáč

Simula Research Laboratory

Walter Tichy

Universität Karlsruhe

Dag I. K. Sjøberg

Simula Research Laboratory

Erik Arisholm

Simula Research Laboratory

Magne Aldrin

Norwegian Computing Center

Abstract

Software "Design Patterns" seek to package proven solutions to design problems in a form that makes it possible to find, adapt and reuse them. To support the industrial use of Design Patterns, this research investigates when, and how, using patterns is beneficial, and whether some patterns are more difficult to use than others. This paper describes a replication of an earlier controlled experiment on Design Patterns in maintenance, with major extensions. Experimental realism was increased by using a real programming environment instead of pen and paper, and paid professionals from multiple major consultancy companies as subjects.

Measurements of elapsed time and correctness were analyzed using regression models and an estimation method that took into account the correlations present in the raw data. Together with on-line logging of the subjects' work, this made possible a better qualitative understanding of the results.

The results indicate quite strongly that some patterns are much easier to understand and use than others. In particular, the VISITOR pattern caused much

confusion. Conversely, the patterns OBSERVER and, to a certain extent, DECORATOR were grasped and used intuitively, even by subjects with little or no knowledge of patterns.

The implication is that Design Patterns are not universally good or bad, but must be used in a way that matches the problem and the people. When approaching a program with documented Design Patterns, even basic training can improve both the speed and quality of maintenance activities.

Key words: Controlled experiment, design patterns, real programming environment, qualitative results

1.1 Introduction

Design Patterns have become quite popular (**??**). In addition to making design knowledge available to both junior and more experienced developers, it is claimed that Design Patterns define a common terminology that can be used to document the design. According to the classic books by (**??**), individual patterns can be combined into a language that guides the designer. This should simplify communication of the underlying design and assumptions from the original designers to maintainers of the software.

An expected benefit—because Design Patterns tend to provide solutions that are more complete than just solving the immediate problem at hand—is the ability to add functionality at a later time without causing major changes. The same property may, however, introduce unneeded complexity.

Prechelt *et al.* performed a controlled experiment in late 1997 (**?**) to measure the effects of using several Design Patterns in a maintenance situation. They used 29 unpaid professionals from a single company as subjects, and used pen and paper for the programming exercises. Based on the properties of the four selected patterns, they hypothesised both positive and negative effects from the patterns.

Their results generally agreed with expectations. The use of the OBSERVER pattern in a simple program had the expected negative effect on maintainability; the VISITOR pattern was neutral in a context

where a negative effect was expected. The DECORATOR pattern had the expected positive effect, and ABSTFACTORY caused only small differences.

The authors of the present paper replicated their experiment with 44 paid, professional subjects using the same programs in a real programming environment, instead of pen and paper. This increases the experimental realism and, thereby, the applicability of the results.

The technical environment also allowed us to collect more data, making more detailed analysis and inferences possible. It also allowed us to address some of the threats to validity of the original experiment, such as effects of subjects' C++ knowledge, and of actually programming and testing the solutions. At the same time, performing a close replication allowed us to do a direct comparison of our results to those of the original experiment.

Our results reinforce the conclusion that each Design Pattern has its own nature and proper place of use; they cannot be classified as "good" or "bad" in general terms. We found a positive effect for OBSERVER and a very strong negative effect for VISITOR, while DECORATOR and ABSTFACTORY found effects similar to those of the original experiment.

The remainder of this paper is organised as follows: Section 1.2 describes the original experiment. The present replication is described in Section 1.3. Section 1.4 contains the programs, work tasks, hypotheses and a summary of the quantitative results. Section 1.5 discusses the results and qualitative factors that underlie the quantitative measurements. Section 1.6 compares this replication to the original experiment. Section 1.7 addresses methods, and Section 1.8 the validity and applicability of the experiment. Section 1.9 concludes.

1.2 The original experiment

This Section gives an overview of the design of the original experiment.

1.2.1 Objectives and hypotheses

"If you have a hammer, everything looks like a nail". Thus, having learned some Design Patterns, it may be tempting for a designer to use them even in situations where their complexity and application may not be warranted, and a simpler solution is available.

Prechelt *et al.* wished to test whether the use of some specific patterns in such situations is "helpful", "harmful" or "neutral" for subjects with different backgrounds. They informally framed their hypotheses as expectations: A design pattern P does, or does not, improve the performance of subjects doing maintenance work task X on program A (containing P) when compared with subjects doing the same work task X on an alternative program A' (not containing P).

Note that the programs may contain patterns other than the one being tested; these other patterns are used identically in the A and A' versions. Henceforth, we will use the term PAT for the version *with* patterns, and ALT for the version *without*.

The "helpful", "harmful" and "neutral" interpretations are derived from the support or contradiction of these hypotheses.

1.2.2 Variables

The experiment used three independent variables:

- **Program and Work Task**: there were four different programs, each with its own purpose, patterns, and two maintenance work tasks.
- **Program Version**: each program existed in two versions, functionally equivalent, comparably complex and sharing some code. The PAT version contained one pattern not present in the ALT version; the alternate version used a "simpler" structure to replace the pattern. This was the central variable of the experiment. Equivalent documentation was present in the two versions, such as an inheritance outline vs. pattern name and class role.
- **Amount of Design Pattern Knowledge**: the experiment was divided into three parts. In the first half of day one, the subjects per-

formed the pre-test, consisting of work tasks on two programs. The rest of day one and the first half of day two contained a patterns course, and the rest of day two was used for the remaining two tasks (post-test).
Before the experiment, most subjects had little or no experience with patterns; thus, the post-test represented subjects with significantly more knowledge of Design Patterns than the pre-test. The experimental design is summarized in Figure 1.1. In order to control learning and fatigue effects, the order of programs is varied, and data is collected from each subject on both PAT and ALT programs.

The experiment used two dependent variables:

- **Time**: the time taken to complete each task, in minutes.
- **Correctness**: each solution was evaluated on a five-point scale to assess to what degree it was functionally correct, regardless of whether it used the "proper" design. We use the term "correctness" instead of the more general "quality", as overall quality is complex and difficult to measure. However, "correctness" is more simply defined:
  1. *Requirements misunderstood*—the solution did not address the given task, or was totally useless, or no real solution was made or attempted.
  2. *Wrong answer*—the requirements were understood, but the attempted solution did not work and was not on the right track.
  3. *Right idea*—the requirements were understood and a reasonable solution attempt was made, but the solution either did not work or did not compile.
  4. *Almost correct*—the solution compiled and ran but did not give exactly the correct answer; however, it did not contain any fundamental errors.
  5. *Correct*—the solution compiled, ran and produced correct output.

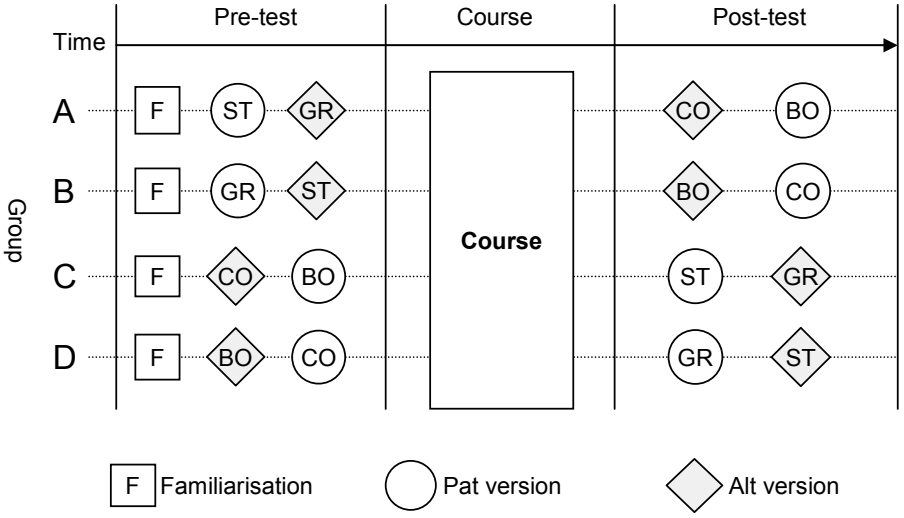

Figure 1.1: Experimental Design: Circles denote PAT program versions, shaded diamonds ALT versions. The two-letter codes are the program name abbreviations. Time runs from left to right; the first day includes the pre-test and the first half of the course, the second day contains the second half of the course and the post-test.

1.2.3 Summary of programs and work tasks

The four C++ programs used came from different domains and were of varying complexity. This was intended to guard against the possibility of domain knowledge or complexity systematically biasing the results; a fuller discussion can be found in Threats to Validity, Section 1.8.

This experiment looked at the effects of Design Patterns through the medium of code (with some documentation), not models. In a maintenance situation, there may not be any valid models available; also, having to actually implement changes in code provides a stricter test of understanding of the original code and its design. Fundamentally, the end result that matters from a development or maintenance project is the final code and not the underlying model.

Throughout this paper, the programs are identified by their names or abbreviations. The names reflect the domain of the programs: Stock Ticker (ST), Graphics Library (GR), Boolean Formulas (BO) and Com-

munication Library (CO). For each program, there were two work tasks. With one exception, the first task was a programming task (addition of a feature), and the second task was oriented more towards theoretical comprehension.

Each program tested a different pattern. While it would be possible to create a program that contained all the patterns, it was considered easier to make the programs separately, so that each tested the aspects of a single pattern. The correspondence between the PAT and ALT versions of the programs was also considered simpler to maintain in separate programs. Most importantly, combining multiple patterns in a single program would have given the subjects a chance to see all of the patterns and code immediately, thereby introducing a learning effect and seriously compromising the experimental design.

In all cases, the features to be added to the programs corresponded to features already present in the code, which could be used as templates by the subjects. Table 1.1 contains short, comparable descriptions of all programs and tasks, while detailed descriptions can be found in Section 1.5.

1.2.4 Subjects, programs, tasks and groups

A total of 29 subjects participated in the original experiment. They were all professional software engineers and came from a single company. Fifteen subjects had some experience with Design Patterns.

The subjects were divided into four groups (A-D). Each group maintained one PAT and one ALT version of a program in both the pre-test and post-test. Each subject worked on all four programs and each program was used as often in the pre-test as in the post-test, and as often in PAT and ALT versions, as shown in Figure 1.1.

1.2.5 Analysis and statistical methods

The time and correctness data was first evaluated using an analysis of variance to identify significant factors. As expected, the work task was the most significant factor, while the order of tasks was not significant.

Table 1.1: Descriptions of programs and tasks

Program	Description and Complexity	Tasks	Patterns
Stock Ticker (ST)	Display an incoming stream of data (read from a file) in one or more windows using a supplied, simple GUI library Simple program with little data and low code complexity PAT: 441 SLOC, 7 classes ALT: 374 SLOC, 7 classes	1: Add another kind of window (the window itself was supplied). 2: Let the user choose which windows should be visible	PAT: OBSERVER ALT: None
Boolean Formulas (BO)	A system for storing and manipulating boolean formulas, represented in a hierarchical data structure. Relatively complex, using a recursive data structure. PAT: 471 SLOC, 11 classes ALT: 372 SLOC, 8 classes	1: Evaluation of formulas 2: Change name of one method, breaking the COMPOSITE pattern	PAT: COMPOSITE, VISITOR ALT: COMPOSITE
Comm. Library (CO)	Wrappers for communication primitives such as transmit, receive, compress and decompress Very little data and not very complex. Simple primitives with similar interfaces PAT: 404 SLOC, 6 classes ALT: 342 SLOC, 1 class	1: Add a wrapper for a new (supplied) primitive 2: Determine the conditions leading to a certain status value; determine how to create a channel with certain functionality	PAT: DECORATOR ALT: None
Graphics Library (GR)	Represent graphic primitives such as point, line and circle, and a system for drawing them on several kinds of device. Methods for creating devices and corresponding primitives. Data structure is partly recursive, but less complex than in "Boolean Formulas". The code is larger than the other programs, but well structured PAT: 683 SLOC, 13 classes ALT: 667 SLOC, 11 classes	1: Add a new graphics device and corresponding subclasses of primitives 2: Determine whether a running supplied method will result in a certain output	PAT: ABSTRACT FACTORY, COMPOSITE ALT: ABSTRACT FACTORY

The rest of the factors (pre/post, pat/alt, individual differences) were discussed on a per-task basis.

Distribution-independent bootstrap methods were used to evaluate mean elapsed times and derive P-values for the differences (**?**). Such differences were calculated for each pair that corresponded to a hypothesis, eg., for PRE-ALT vs. POST-ALT for a particular program and work task. Numerous significant differences were found and compared with the expected trends (hypotheses) for each work task. For many tasks, all groups achieved near-perfect correctness, so the dependent variable "correctness" was often ignored (**?**), p. 1136.

1.3 Current replication

We wished to increase the realism of the experiment (**?**), and attempted to do so in two ways: 1) we used a real programming environment, instead of annotations to paper printouts, and 2) our subjects came from multiple consultancy companies and were paid for their participation, instead of being volunteers from a single company.

We used the same general design of the experiment and the exact same set of four tasks, with a PAT and ALT program version of each. The program code was identical to that used in the original experiment, except for corrections of minor errors. The same course on Design Patterns was taught by the same person (Walter Tichy), using the same course materials.

In the terminology of Lindsay and Ehrenberg, this can be considered a relatively "Close" replication (**?**). While an *identical* replication is neither possible nor particularly desirable, we designed ours to keep it as close as possible, except for differences that are either unavoidable or explicitly desired.

In this case, the difference in actual subjects and their nationality was unavoidable. Their background was roughly the same and was evaluated using the same methods. The use of paid subjects from more than one company, and the use of a programming environment, were motivated by the increased realism they offer.

1.3.1 Logging and data collection

Our subjects used their own laptop PC's as terminals, while the actual programming environment ran on a set of Windows Terminal Servers. This made it possible to install various non-intrusive logging tools to collect additional data, beyond the correctness and time variables (as well as the post-mortem questionnaire) of the original design.

To gain insight into the programming process of each subject, a copy of the program file being edited was saved at every compilation, together with information on compilation errors, editing time (as a further check) and breakpoint and debugging information. It is, therefore, possible to detect the changes made for each compilation, which often occurs every few minutes, as well as the debugging method used. This data was used both for grading of solution correctness (Section 1.4.2) and in the qualitative analysis of results (Section 1.5).

To limit the possibilities for cheating,and to lessen competitive stress, subjects were placed so that two people sitting next to each other always worked on different programs. Copy/Paste operations through the Terminal Server environment need multiple menu choices and intermediate files, making data exchange through IR ports impractical. Inspection of the code logs revealed no traces of cheating or plagiarism.

1.3.2 Subject selection and background

Several international and Norwegian consultancy companies contributed subjects. They were explicitly asked to provide people who formed a reasonably representative sample, with regard to seniority, experience and education.

In total, 44 subjects were paid for their participation, on three scales (junior, intermediate, senior). The employers determined the scale for each participant. Additionally, payment was offered for a limited amount of overhead per company, to encourage them to undertake a serious selection process.

The subjects were mostly (39) professional software engineers, from 11 different companies. There were also five students at the master/PhD

level. The median education was five years and work experience was four years. Five subjects had 20 or more years work experience. 17 subjects had some experience with patterns, though generally with only one or a few patterns, applied a few times. Only six subjects had practical knowledge of the patterns actually being tested.

Regarding prior experience with object-oriented programming and C++, one third of the subjects answered that they had less than one year experience with object-oriented programming (as opposed to other paradigms); the average value was 2.4 years. 75% of the subjects had written less than 25 000 lines of C++ code.

Thus, the participants in our replication generally had a relatively extensive education, but only limited practical experience, and initially almost no relevant pattern knowledge. We would expect the lack of practical experience to cause the subjects to spend more time on some programming details than would experienced developers. This also has some implications for the external validity of the experiment.

1.3.3 Group assignment

The subjects were assigned to the four groups using randomized blocking, where the groups were balanced (the blocks were not random, but the assignment of members from each block to the groups was). Obviously, balancing all characteristics at once is not possible; the greatest weight was given to knowledge of Design Patterns, and general experience. The subjects completed a survey form before the experiment, and their answers were used to compute a "pre-qualification score". The subjects were ordered by this score, and those with the four highest scores were randomly assigned, one to each group, then the next four, etc.

Of the original 54 subjects who expressed an interest, 10 were unable to participate. Of those, four cancelled after the final group assignment, causing an imbalance in group sizes. Table 1.2 summarizes the groups. Possible threats to validity stemming from the imbalances are discussed in Section 1.7.8.

Table 1.2: Subject backgrounds for each of the four groups A–D

	N	Pat	Educ	Work	OO prog	C++
A	10	1	3.5	5.7	2.2	13300
B	12	1	4.7	5.3	2.6	16387
C	12	2	4.0	6.7	1.5	8509
D	10	2	4.1	7.7	3.4	18638

N = number of subjects,

Pat = number with previous knowledge of relevant patterns,

Educ = median education (years),

Work = median work experience (years),

OO prog = median OO programming experience (years),

C++ = mean number of C++ lines of code written.

1.3.4 Experiment conduct

The subjects were *not* told about the design of the experiment (the presence of Pattern and Alternate versions), nor about what we were measuring, logging or how this was done. The programming tasks were presented as exercises for the patterns course, though the subjects were told in advance that they were taking part in a combination of course and experiment. The authors discreetly eavesdropped on conversations during lunch and in breaks, and the subjects did not to our knowledge discuss the tasks.

The same general timetable was followed as in the original experiment: The pre-test work tasks in the morning, then lunch, followed by the first part of the Design Patterns course. On day two, the course continued until lunch, and the post-test work tasks were conducted after lunch.

The participants were encouraged to work until they were done. On day one there was a time limit (the start of the course), on day two there was no formal time limit and the last subject left at 5.45 pm. Four subjects ran out of time on day one, and in the questionnaire estimated that they would have needed from one to three hours additional time to complete their tasks. Since our analysis looks at both the time needed

for the tasks as well as the solution quality, these partial solutions were graded and included in the data set. The analysis of elapsed time excluded solutions of low quality, regardless of the reason for the low quality (Section 1.3.6).

1.3.5 Expectations and hypotheses

Since both the programs and the different patterns they contained were of varying kind and complexity, the hypotheses varied. In some cases we expected the PAT version to be easier to understand and modify, while in other cases we expected the ALT version to have the advantage. The expected effect of the patterns course also differed.

The hypotheses represent what we expected to observe based on software engineering common sense. They were identical to those of the original experiment, which used bootstrap methods to compare mean work times for work tasks, either between PAT and ALT versions or between PRE and POST. The hypotheses were defined and evaluated separately for each program and work task.We reformulated the hypotheses to correspond to our statistical approach, and they are presented in tabular form in Section 1.3.8, table 1.3.

In addition to quantitative analysis of dependent variables, a qualitative analysis was also made, the purpose of the latter being to try to explain *why* the quantitative results were observed.

1.3.6 Model for analysis of time

To evaluate the observed quantitative data and enable a more compact representation of the hypotheses, a regression-based approach was adopted. The method used in the original experiment (bootstrap estimations of distributions of differences of means) only takes into account data for each pair of tasks considered, separately from all other data. The model adopted here considers all the data simultaneously and thereby enables us to better take into account differences between individual subjects.

Since completion times have little meaning for solutions with low cor-

rectness, only those solutions achieving correctness score 4 ("Almost correct") or 5 ("Correct") were used in this analysis.

The time used to execute a task may vary systematically by explanatory variables such as program and task number, ALT or PAT version, and amount of pattern knowledge. Define

$time_{t,i} =$ time used by individual i ($i = 1, \ldots, n$) on task t ($t = 1, \ldots, 8$), on the condition that the corresponding solution correctness was at least 4.

$I_{P,t,i} = 1$ if task t for individual i were with PAT, else $I_{P,t,i} = 0$,

$I_{C,t,i} = 1$ if task t for individual i were done after the course, else $I_{C,t,i} = 0$,

Further, let $\mathrm{E}(time_{t,i}) = \mu_{t,i}$ be the expected time used on task t, where the expectation is taken over the sample population of programmers, given specific values of the explanatory variables. We assume that the logarithm of $\mu_{t,i}$ has the additive structure

$$\log(\mu_{t,i}) = \alpha_t + \beta_t I_{P,t,i} + \delta_t (1 - I_{P,t,i}) I_{C,t,i} + \gamma_t I_{P,t,i} I_{C,t,i} \qquad (1.1)$$

where the α's, β's, δ's and γ's are regression coefficients that will be estimated from the data.

The model can be transformed back to the original scale. The population averaged expected time used on task t before the course, for ALT programs, and with correctness at least 4 will be $\mu_{t,i} = b_t = \exp(\alpha_t)$.

The quantity b_t will be called the *base level* for task t. The expected time for a PAT program, before the course, and with correctness at least 4, will be $\mu_{t,i} = b_t \cdot \exp(\beta_t)$, such that

$\exp(\beta_t)$ is the relative increase in time by using PAT instead of ALT before the course, i.e., the "effect of Design Patterns before course".

Furthermore, the following quantities will be of interest:

$\exp(\delta_t)$ is the relative increase in time from PRE-ALT to POST-ALT, i.e., the "course effect on alternate programs",

$\exp(\gamma_t)$ is the relative increase in time from PRE-PAT to POST-PAT, i.e., the "course effect on Design Patterns programs",

$\exp(\beta_t + \gamma_t - \delta_t)$ is the relative increase in time by using PAT instead of ALT after the course, i.e., the "effect of Design Patterns after course".

The relative increases will be reported as percentage increases, i.e., instead of reporting $\exp(\beta_t)$, we will report $100 \cdot \exp(\beta_t) - 100$, etc.

If we assume that the observations $time_{t,i}$ are gamma distributed, and independent for all t and i, the parameters can be estimated by maximum likelihood according to the theory of generalized linear models (GLM) (**?**). The gamma distribution is suitable for data that takes only positive values and are skewed to the right. This is the case for the *time* data, which has 0 as lower limit, but no clear upper limit (though it cannot be longer than a day). However, the independence assumption is unrealistic, as we have multiple observations for each individual subject, one for each work task.

Therefore, the parameters were instead estimated by the method of Generalized Estimating Equations (GEE) (**??**), using the software package Oswald (**?**). GEE is an extension of GLM, developed specifically to accommodate data that is correlated within clusters (here individuals).

First, the user has to specify a so-called working correlation matrix, i.e., the structure of the correlations between observations within the same individuals. For the present model, we have used an "exchangeable correlation matrix", which means that all observations within the same individual have equal correlation. Then the estimation is carried out under the assumption that (the structure of) the working correlation matrix is true, and standard errors of the estimates are calculated.

The theory of GEE states that the estimates are asymptotically normal distributed with the given standard errors. Further, under certain assumptions, the estimates are consistent (i.e., converge to the true values when the number of observations becomes large), *even if the distribution or the working correlation matrix is incorrectly specified*. This important result does not imply that the choices of distribution and working correlation matrix are of no consequence. The closer they are to reality, the more precise the estimates will be.

1.3.7 Model for analysis of correctness

It would be natural to handle the correctness scores by ordinal logistic regression, i.e., by estimating the probabilities of getting the score values $1, 2, .., 5$, given the explanatory variables. However, it was impossible to estimate such a model by GEE or GLM, because the methods break down when all observations for certain combinations of the explanatory variables have the same value. This happened in several cases; for example, in the PAT group working on task 1 of the Stock Ticker program in the post-test, all the subjects had a perfect score. Instead we have used the model presented below, assuming Gaussian data.

The model for the quality or correctness score on each task is similar to that for time. Define $score_{t,i} \in (1, 2, 3, 4, 5)$ = score achieved by individual i ($i = 1, \ldots, n$) on task t ($t = 1, \ldots, 8$), .

Further, let $\mathrm{E}(score_{t,i}) = \mu_{t,i}$ be the expected score on task t, which is assumed to have the structure

$$\log(\mu_{t,i}) = \alpha_t + \beta_t I_{P,t,i} + \delta_t (1 - I_{P,t,i}) I_{C,t,i} + \gamma_t I_{P,t,i} I_{C,t,i}. \qquad (1.2)$$

The regression coefficients have different values than in the time model, and slightly different interpretations. The expected score for task t before the course, for ALT programs now becomes $\mu_{t,i} = \alpha_t$, where α_t will be called the base level for task t. The other coefficients give the increase in score compared with the same alternatives as in the time model. Note that *positive* values here mean improvements in correctness (higher correctness score), whereas *negative* values meant improvements in the time model (shorter time).

The parameters have again been estimated by GEE, but now using the Gaussian family of distribution as mentioned above. We used an "identity working correlation matrix", because the GEE algorithm did not converge with an exchangeable working correlation matrix. Using an identical working correlation matrix gives the same estimates as GLM, but the estimated uncertainty limits are more robust to incorrect specification of the correlation.

In practice, the correctness scores take integer values, and are far from

Table 1.3: Hypotheses for time, legend on p. 65

Pr	Task	POST-ALT vs. PRE-ALT	POST-PAT vs. PRE-PAT	PRE-PAT vs. PRE-ALT	POST-PAT vs. POST-ALT
ST	1			S1 +	S2 −
	2				S3 −
BO	1	B2a −	B2b −	B1 +	B3 −
	2			B4 +	B6 0
CO	1			C1 −	C2 −
	2			C3a +	C3b +
GR	1	G2a −	G2b −	G1 +	
	2	G4a −	G4b −	G3a 0	G3b 0

Gaussian. As mentioned in the discussion of the *time* model, the GEE estimates are robust also to mis-specification of the distribution. However, more data would be necessary for the asymptotics to hold, so the uncertainty limits should be interpreted with some care.

To guard against a model optimized to find only the "desired" results and ensure its statistical correctness, it was constructed by one of the authors (M.A.) without prior detailed knowledge of the hypotheses posed.

1.3.8 Reformulated hypotheses

Given the analysis models and the quantities $\exp(\beta_t), \exp(\gamma_t), \exp(\beta_t + \gamma_t - \delta_t)$, we can now express the hypotheses formally, in a tabular format. In Tables 1.3 and 1.4, a '+' in a cell means that we expected a positive value for the coefficient on the log scale or greater than 1 on the original scale (longer time, higher correctness score). A '−' means we expected a negative coefficient on the log scale or lower than 1 on the original scale. A '0' means we expected no change relative to the base level (log scale coefficient 0, original scale 1), which is the ALT version of each program, before the patterns course. The hypotheses and expectations are discussed in detail in Section 1.5.

Table 1.4: Hypotheses for correctness

Pr	Task	POST-ALT vs. PRE-ALT	POST-PAT vs. PRE-PAT	PRE-PAT vs. PRE-ALT	POST-PAT vs. POST-ALT
ST	1				
	2				
BO	1				
	2			B5 –	
CO	1				
	2			C4a –	C4b –
GR	1				
	2				

Note that empty cells in this table mean that no hypothesis was advanced with respect to this program/task/parameter combination. As this is a replication of an earlier experiment, we did not change any hypotheses or advance any new ones. Significant observations that do not correspond to one of the hypotheses are discussed in Section 1.5.6. The numbering of the hypotheses was rendered consistent with those in the original experiment, to make comparisons easier.

The headings refer to the effect measured: POST-ALT vs. PRE-ALT shows the course effect (from pre-test to post-test) on the ALT version programs, while PRE-PAT vs. PRE-ALT shows the effect of going from ALT to PAT version, both taken in the pre-test only.

1.4 Results

1.4.1 Validation of raw data

The first step in the analysis was to check that there were no errors in the raw data resulting from misunderstandings or gross technical problems. The only such case was one subject who had performed the work tasks completely out of order. All data from this subject was therefore dropped.

1.4.2 Grading of correctness

The grading of the solution correctness used the scale described in Section 1.2.2 above, ranging from "Misunderstood" to "Correct".Correctness was determined by first compiling and running the final solution saved by each subject. Then, the final solution code was inspected to determine the magnitude of any problems. Finally, all intermediate source files were inspected to arrive at a better understanding of any errors and the solution strategy. The grading was done by one of the authors using a system that presented the source files, program output, etc. The subject information was fully anonymised at this point (to the grader) and the subjects were graded in random order.

We also determined whether each solution used the patterns present in the code. Note that "Correct" does not imply that the patterns present in the code, if any, were *actually used in the solution*; only that the solution produced the correct output.

Four subjects had consistently low-quality solutions. Inspection on a per-compilation basis revealed that their C++ proficiency was so low that it would significantly mask any other effect. None of them finished all tasks, and most had given up (i.e., stopped work while the solution was nonworking and there was more time available) on more than one task. All data from these subjects was also dropped. Since their solution correctness was consistently low across both PAT and ALT program versions, this introduces no significant bias.

1.4.3 Refinement of the analysis model

There were several candidates for explanatory variables other than those described in Sections 1.3.6 and 1.3.7. In the model for dependent variable *time*, candidate explanatory variables were the pre-qualification score and the correctness of the solutions. The pre-qualification score (as discussed in Section 1.3.3) should be significant if it is correlated with the actual performance of the subjects. An analysis of the data showed that this was *not* the case. The coefficient had a value close to 0 and was not significant.

We interpret this as showing that the pre-qualification score bears little relation to the subjects' actual performance. At the same time, the experimental design is quite robust with respect to effects of individual performance differences, and therefore also balancing of groups. Since all the subjects performed tasks on all the programs, imbalances between subjects and groups will increase the total variability, but have a relatively small chance of causing systematic skewing of the results.

Solution correctness could also have been included in the analysis model for time, in the form of an indicator variable $I_{Q,t,i}$ with value 1 if solution correctness 5 were achieved by a subject on a task. One might expect a positive value for this coefficient, indicating that achieving higher correctness takes more time. An analysis showed that it was also close to 0 and not significant.

Our interpretation is that high correctness was not achieved at the expense of time; i.e., skilled individuals tend to favour time and correctness equally.

In the model for dependent variable *correctness*, the pre-qualification score had a low, positive value ($p = 0.026$), but the values and confidence intervals of the other estimated coefficients in the model were not significantly changed by including this factor.

To summarize, including these candidate explanatory variables caused only very slight changes to the values and confidence intervals of the remaining coefficients, in the models for both time and correctness. They were therefore not included in the final model for time.

1.4.4 Effect of programming tool use

We wished to determine whether any subjects had spent a significant amount of time on "technical details", i.e., problems with programming language syntax, obscure compiler error messages or other factors that might be classed as not relevant to the effects of design patterns. This determination was necessarily exploratory in nature and proceeded as follows:

An analysis was performed of each separate compilation of each solu-

tion. A "syntactical change" was defined as one that did not introduce new features or functions, but only changed the statement that caused the compilation errors. Typically this consisted of trying out various combinations of the `.`, `->`, `::` or `*` operators, different placements of `[]` brackets in attempted array declarations, etc. Another example was a subject who spent about 15 minutes looking for a missing closing brace; the error messages from the compiler were not helpful.

If the program submitted for compilation did not compile, the only changes were localized and syntactical, and there was a contiguous series of such changes all related to one or a few lines, those individual compilations were classified as "irrelevant technical detail". The sum of the editing times for such compilations was subtracted from the total elapsed time for that task, as a correction.

Such corrections[2] were made for 33 out of the 43 subjects. The regression analysis was then run on both corrected and uncorrected data, to check whether the corrections actually had any effect, and to guard against the introduction of any bias. Ideally, we would want the confidence intervals to shrink when using the corrections, though without significantly changing the point estimates.

The corrections did achieve some reduction in the confidence intervals, and did so without materially affecting the point estimates. However, the reduction was nowhere near significant and did not change the degree of support or rejection for any hypothesis. Since the grading that underlies such corrections is necessarily somewhat subjective, and there is a risk of penalizing subjects who simply spent time thinking about the problems without submitting compilations, the corrections were dropped and the final analysis done on *uncorrected, raw data*.

---

2. All data is stored in a relational database together with the relevant source files, so that it is possible at any time to retrieve data with or without any corrections and grades, inspect the classification of each compilation, and the file difference giving rise to it.

1.4.5 Summary of quantitative results

The results from the analysis using the regression models are shown in graphically in Figures 1.2 and 1.3, and in tabular form in Table 1.7 using the same layout as for the hypotheses in Table 1.3.

In the Figures, the point estimates for the coefficients are dots and 95% confidence limits are shown as vertical bars; a significant (at 5%) result is one where the bars do not cross the 0 line. As detailed in Section 1.3.6, the estimates from the regression model are asymptotically normal distributed, providing the basis for calculation of the confidence intervals.

Descriptive statistics are given in Table 1.5 for working times and Table 1.6 for correctness scores. The first four columns in the tables contain the key to the measurement—the program, work task, PAT/ALT version, and pretest/posttest.

From Table 1.7 we can see that significant results (at 5%) were achieved for five out of the total 20 hypotheses, while another seven tests showed a reasonably certain direction, either supporting or contradicting the hypothesis.

The regression model provides strong support for the hypotheses in four cases:

1. Using the VISITOR pattern causes problems if the underlying data structure changes
2. DECORATOR is a pattern that requires training, but then yields easier maintenance
3. DECORATOR makes it more difficult to trace the flow of control in a program, and increases the time needed to understand it
4. Like DECORATOR, OBSERVER requires some training, but is then easy to understand and shortens maintenance

The hypothesis that a short course is sufficient to profit from ABSTRACT FACTORY was strongly contradicted. The observed result was actually the opposite; subjects took significantly longer after the course than before to complete the task.

Line	Prog	Task	Pre/Post	Ver	Min	Q1	Mean	Q3	Max	N
1	BO	1	Pre	Alt	27	88	129	175	298	9
2	BO	1	Post	Alt	27	45	86	130	186	9
3	BO	2	Pre	Alt	0	4	13	20	37	6
4	BO	2	Post	Alt	4	8	14	17	39	8
5	BO	1	Pre	Pat	45	86	108	135	145	11
6	BO	1	Post	Pat	20	55	99	160	173	10
7	BO	2	Pre	Pat	6	13	26	39	66	8
8	BO	2	Post	Pat	1	3	24	46	65	6
9	CO	1	Pre	Alt	22	35	63	85	97	10
10	CO	1	Post	Alt	24	48	69	90	117	10
11	CO	2	Pre	Alt	7	9	15	20	33	10
12	CO	2	Post	Alt	7	9	19	27	47	10
13	CO	1	Pre	Pat	44	45	65	84	124	8
14	CO	1	Post	Pat	17	22	33	43	60	9
15	CO	2	Pre	Pat	12	18	33	45	66	8
16	CO	2	Post	Pat	7	11	20	29	45	9
17	GR	1	Pre	Alt	32	51	91	136	196	10
18	GR	1	Post	Alt	37	57	105	155	214	11
19	GR	2	Pre	Alt	13	16	37	51	125	8
20	GR	2	Post	Alt	11	22	41	43	122	11
21	GR	1	Pre	Pat	36	59	78	89	155	9
22	GR	1	Post	Pat	101	104	127	145	190	9
23	GR	2	Pre	Pat	9	14	28	42	65	9
24	GR	2	Post	Pat	6	9	24	38	50	9
25	ST	1	Pre	Alt	6	7	14	21	32	9
26	ST	1	Post	Alt	7	13	26	35	65	9
27	ST	2	Pre	Alt	12	15	31	48	68	9
28	ST	2	Post	Alt	2	23	50	77	143	9
29	ST	1	Pre	Pat	1	3	22	27	85	10
30	ST	1	Post	Pat	2	6	14	26	40	11
31	ST	2	Pre	Pat	1	14	36	56	91	9
32	ST	2	Post	Pat	5	7	15	22	36	11

Table 1.5: Descriptive statistics for time: Each line contains the minimum, first quartile, mean, third quartile and maximum values of programming time in minutes, for one combination of Program, ALT or PAT version, task number and day number. Number of subjects is also given.
Each hypothesis in table 1.3 refers to one pair of lines in this table, e.g., hypothesis (B2a: $-$) compares BO/ALT/Task 1/POST: line 2 to BO/ALT/Task 1/PRE: line 1, and expects the former to be lower (shorter time).

Line	Prog	Task	Pre/Post	Ver	Min	Q1	Mean	Q3	Max	N
1	BO	1	Pre	Alt	1	3.0	3.9	5.0	5	9
2	BO	1	Post	Alt	3	4.0	4.6	5.0	5	9
3	BO	2	Pre	Alt	2	2.8	4.2	5.0	5	6
4	BO	2	Post	Alt	1	2.0	3.8	5.0	5	8
5	BO	1	Pre	Pat	2	2.0	3.5	5.0	5	11
6	BO	1	Post	Pat	1	1.8	3.0	5.0	5	10
7	BO	2	Pre	Pat	2	2.3	3.6	5.0	5	8
8	BO	2	Post	Pat	2	2.0	2.8	4.3	5	6
9	CO	1	Pre	Alt	3	3.8	4.5	5.0	5	10
10	CO	1	Post	Alt	3	3.8	4.4	5.0	5	10
11	CO	2	Pre	Alt	3	3.8	4.4	5.0	5	10
12	CO	2	Post	Alt	5	5.0	5.0	5.0	5	10
13	CO	1	Pre	Pat	5	5.0	5.0	5.0	5	8
14	CO	1	Post	Pat	5	5.0	5.0	5.0	5	9
15	CO	2	Pre	Pat	4	5.0	4.9	5.0	5	8
16	CO	2	Post	Pat	4	5.0	4.9	5.0	5	9
17	GR	1	Pre	Alt	1	3.0	3.7	5.0	5	10
18	GR	1	Post	Alt	3	5.0	4.7	5.0	5	11
19	GR	2	Pre	Alt	2	2.0	3.9	5.0	5	8
20	GR	2	Post	Alt	2	4.0	4.5	5.0	5	11
21	GR	1	Pre	Pat	5	5.0	5.0	5.0	5	9
22	GR	1	Post	Pat	3	4.5	4.7	5.0	5	9
23	GR	2	Pre	Pat	2	2.0	3.9	5.0	5	9
24	GR	2	Post	Pat	2	2.0	3.2	5.0	5	9
25	ST	1	Pre	Alt	4	5.0	4.9	5.0	5	9
26	ST	1	Post	Alt	4	5.0	4.9	5.0	5	9
27	ST	2	Pre	Alt	4	4.0	4.4	5.0	5	9
28	ST	2	Post	Alt	3	4.0	4.4	5.0	5	9
29	ST	1	Pre	Pat	1	4.8	4.5	5.0	5	10
30	ST	1	Post	Pat	5	5.0	5.0	5.0	5	11
31	ST	2	Pre	Pat	5	5.0	5.0	5.0	5	9
32	ST	2	Post	Pat	4	5.0	4.9	5.0	5	11

Table 1.6: Descriptive statistics for quality: Each line contains the minimum, first quartile, mean, third quartile and maximum values of the correctness score, for one combination of Program, ALT or PAT version, task number and day number. Number of subjects is also given.
Each hypothesis in table 1.4 refers to one pair of lines in this table.

Table 1.7: Summary of quantitative results—work time

Pr	Task	Post-Alt vs. Pre-Alt			Post-Pat vs. Pre-Pat			Pre-Pat vs. Pre-Alt			Post-Pat vs. Post-Alt		
ST	1							S1	S	+52%	S2	**SS**	−48%
	2										S3	**SS**	−72%
BO	1	B2a	**S**	−33%	B2b	—		B1	WC	−17%	B3	WC	+29%
	2							B4	S	+108%	B6	WS	18%
CO	1							C1	WC	+13%	C2	**SS**	−49%
	2							C3a	**SS**	+117%	C3b	—	
GR	1	G2a	WC	+2%	G2b	**SC**	+62%	G1	—				
	2	G4a	C	+66%	G4b	—		G3a	S	−9%	G3b	C	−39%

Cell contents: To the left is the program/hypothesis identification, followed by the degree of support in the centre. **SS** means Strongly Supported, **SC** means Strongly Contradicted (significant at 5% level). **S** means supported, **C** means contradicted (not strictly significant at 5%, but still relatively clear effect). **WC** means weakly contradicted, similarly, **WS** denotes weak support. "**—**" denotes an inconclusive result.
The estimated effect in % of the factor on the observed time (multiplicative) is given to the right.

Table 1.8: Summary of quantitative results—correctness

Pr	Task	Post-Alt vs. Pre-Alt	Post-Pat vs. Pre-Pat	Pre-Pat vs. Pre-Alt			Post-Pat vs. Post-Alt		
ST	1								
	2								
BO	1								
	2			B5	S	−20%			
CO	1								
	2			C4a	C	+15%	C4b	WS	−5%
GR	1								
	2								

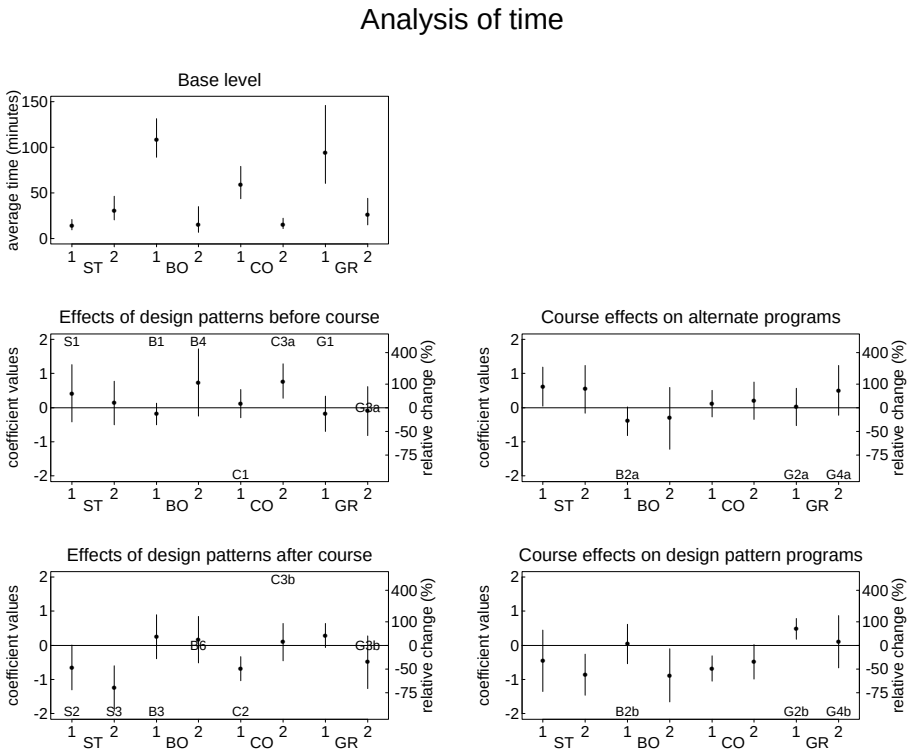

Figure 1.2: Analysis of elapsed times for all programs and tasks
The upper left panel of the Figure shows the base levels b_t for each task, given in minutes. The estimates are given as dots, whereas the vertical lines are 95% confidence intervals. The four lower panels have the same structure as the upper left panel, *but show changes relative to the Base level.* Hypotheses are shown, with the position of the label indicating the direction expected effect. The log scale is given at the left of the panel, and the corresponding relative change in % is given at the right side of the panel.

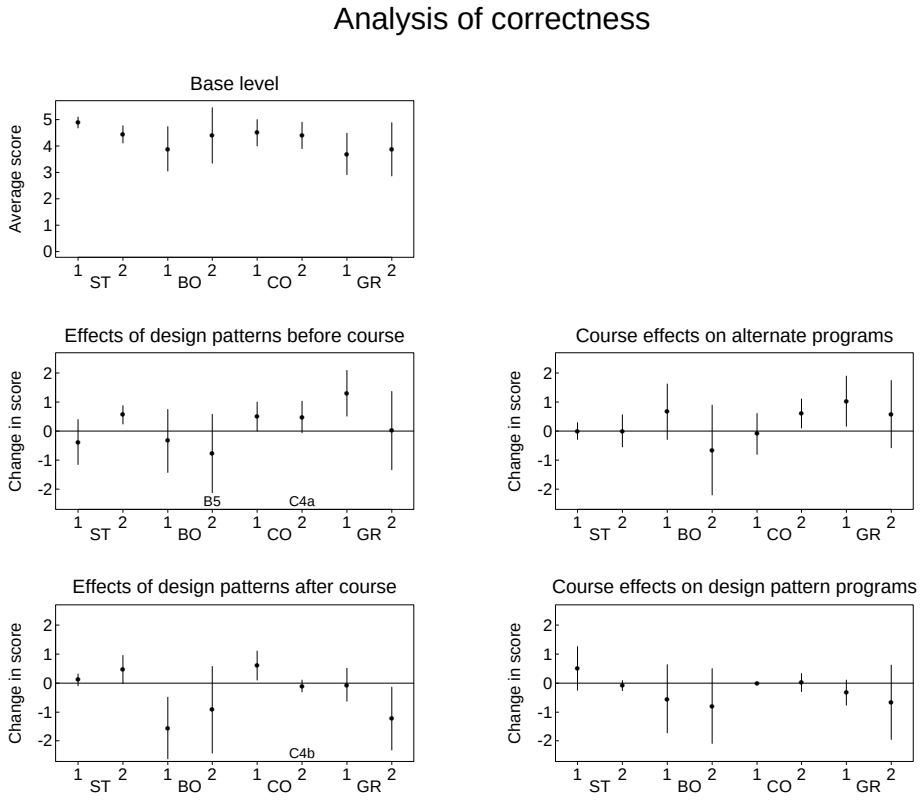

Figure 1.3: Analysis of correctness effects for all programs and tasks
The upper left panel of the Figure shows the base levels α_t for each task, given as average scores. The four lower panels show changes in scores compared with the relevant alternative. Some confidence intervals have 0 length. For the corresponding estimates, all relevant data had the same correctness, or the same change in correctness. In such cases, the GEE method is unable to compute confidence intervals.

1.5 Discussion

The quantitative results and hypothesis tests form only one part of the total results from the experiment, and do not automatically result in better understanding. The quantitative measures must be complemented with qualitative evaluations. Logging all compilations provides a basis for such evaluations, and allows us to gain insight into what happened, and from there to formulate explanations of why.

This discussion is structured around each program and work task, since their kinds and patterns were different, and revealed different aspects of patterns and their effects.

1.5.1 Observer: Stock Ticker (ST)

Stock Ticker is a program for directing continuous streams of data (stock trades) to one or more displays that are part of the program.

Both versions of ST consist of seven classes, and in the PAT version (441 lines) four of them participate in an OBSERVER. The ALT version includes one class that contains an instance variable for each display, and updates the displays when data changes. There is no dynamic registration of observers in this version, which has 374 lines. The line counts include blank lines and comments.

The actual displays are implemented using a very simple `Window` interface. The subjects did not use this interface directly, because the display objects required by the work tasks were already present in the code.

Work task 1

*"In the given program, only one of the three display types is used. Enhance the program such that a second display (of type **volume**) is shown".* The PAT groups only had to invoke the pattern method `subscribe()` with a new instance of the display. The ALT groups had to introduce an instance variable for the new window and invoke its `update()` method to show new data. The main work in this program is to understand how it operates, because the actual changes required are

very small in both cases.

Hypotheses: In the pretest, we expected the PAT group to need more time (S1: +), since subjects without pattern knowledge need to analyze the OBSERVER to determine how it operates. After the course, we expected the opposite (S2: −), because the OBSERVER should then have been easy to grasp and use.

Results: S1:+ ▷ $+52\%^{+253}_{-34}$ is weakly supported.[3] PRE-PAT subjects did use 52% more time than PRE-ALT. Most of the difference can be attributed to one outlier data point; without it, the PRE-PAT and PRE-ALT groups are virtually identical with respect to time used. The correctness is also consistently high for both groups.

The outlier is a subject who did not understand the OBSERVER pattern, and actually reinvented and reimplemented an equivalent structure, which explains why it took so long to complete the task.

Hypothesis S2:− ▷ $-48\%^{0}_{-73}$ is quite strongly supported. There were no significant differences in the correctness of the solutions.

Work task 2

"Change the program so that displays can be dynamically selected at runtime". The code included a third display type for this purpose, as well as a simple method to get user input: `bool askYesNo( char *prompt );`

The PAT groups needed to do very little; just ask the user two or three questions, and `subscribe()` to those displays that were selected. The ALT groups would have to add a mechanism for extending the number of displays with more instance variables.

Hypotheses: In contrast to the original pen and paper experiment, where Prechelt *et al.* stated that the PAT groups did not need to do anything (**?**), the subjects in this experiment actually had to implement

3. Notation: The hypothesis is identified by its letter/number combination as in Tables 1.3, 1.4, 1.7 and 1.8, followed by the direction of the expected effect, and observed effect in %. The lower and upper limits of the 95% confidence interval are given as subscript and superscript.

a change. We did, however, expect the PAT groups to have a clear advantage (S3: $-$), because their changes are much smaller; apart from getting user input, only a test around each `subscribe()` method invocation is needed.

Results: S3:$- \triangleright -72\%^{-45}_{-85}$ is strongly supported. The POST-PAT group spent less than half the time used by the POST-ALT group. Most subjects in the POST-PAT group chose to add Boolean variables to the `TradeInfo` class constructor to specify the users' choice, while a few put the subscribe calls into the main program.

All the subjects in the ALT groups interpreted the task to mean a choice between a fixed number of displays: those already present in the code. None implemented a fully dynamic solution that would easily have accommodated another display.

Correctness was significantly higher for the PAT groups than for the ALT groups both before and after the course. The effect of the course was in this case to reduce the time needed to arrive at a high-correctness solution. OBSERVER, therefore, seems to be a pattern that can be grasped without much training, but training saves time.

1.5.2 Composite and Visitor: Boolean Formulas (BO)

Boolean Formulas contains a library for representing Boolean formulas (`OR`, `AND`, `XOR`, `NOT` and variables), and methods for printing the formulas in two different styles. It also contains a small main program that sets a few variables, constructs a formula and prints it using both methods.

The PAT version consists of 11 classes over 471 lines. The formulas are represented using COMPOSITE, and the printing methods use VISITORs. For each concrete COMPOSITE class there is a printing method in each of the VISITORs, and each COMPOSITE class provides a dispatch method for the VISITOR. Internally, the COMPOSITEs use three different data structures: `NOT` has a single operand, `XOR` has two operands in a classic left-right scheme, while `AND` and `OR` are implemented with a common base class and have a dynamic number of operands to handle expressions such as `a AND b AND c` (this would be one `AND` with

three operands). Recursion is a central feature of the COMPOSITE pattern. The VISITOR solution allows the addition of new functions without changing the COMPOSITEs.

The ALT version has the same COMPOSITEs, but is shorter, with eight classes over 372 lines. The VISITOR is completely missing, and the printing functionality is implemented directly as methods in each concrete COMPOSITE, so adding a new function means adding methods to each concrete COMPOSITE.

Work task 1

"Enhance the program to evaluate Boolean formulas, i.e., to determine the result for a given formula represented by a Composite and values of the variables".

The printing methods serve as structural examples. The PAT groups had to create a new VISITOR, while the ALT groups had to add new methods to each concrete COMPOSITE class.

Hypotheses: In principle, it should be easier to create a single new class similar to another existing class, rather than having to add methods to several classes. This should favour the PAT group.

However, VISITOR is quite a difficult pattern to comprehend and use, so we expected that the PAT group would need more time to understand the structure than the ALT group would need to simply add methods (B1: $+$).

Gaining patterns knowledge during the course should help both groups, since there is a COMPOSITE in both the PAT and ALT versions of the program (B2a: $-$), (B2b: $-$). The PAT group should get an additional advantage from the VISITOR pattern after the course (B3: $-$).

Results: B1:$+ \triangleright -17\%^{+14}_{-40}$ was not supported; the PRE-PAT group actually needed 17% less time than the PRE-ALT group, though much of the difference was due to one outlier.

A more interesting observation comes from the correctness model in Table 1.8 (VISITOR). First, the correctness was quite low, and lower

for the PAT group before the course. Second, the ALT group benefited from the course, while the PAT group actually got worse. Perhaps most interesting is that only three out of 12 subjects in the PRE-PAT group *actually used* the VISITOR. The rest implemented changes directly on the COMPOSITE and ignored the two VISITORs in the code.

B2a:$-\triangleright -33\%\,^{+2}_{-56}$ and B2b:$-\triangleright +4\%\,^{+86}_{-42}$ had some support, most for B2a. There was, however, some rise in correctness for the ALT group.

B3:$-\triangleright +29\%\,^{+144}_{-32}$ was inconclusive, with no significant difference visible. However, the correctness of the PAT group solutions was at the same low level as in the pre-test, and the subjects were *still* not using VISITOR much—four out of 10, and of those four, only one succeeded.

Inspection of the solutions on a compilation-by-compilation basis revealed that many subjects struggled with the recursion inherent in the COMPOSITE. This is somewhat surprising, given that the subjects were professional developers, many employed by major consultancy corporations.

The conclusion is that VISITOR was so difficult that even after a course that gave the instructor excellent feedback (grade better than 4 out 5), most subjects either ignored it or were confused by it.

We may also speculate that developers nowadays use predefined container classes so much that recursion is simply not used on a daily basis any more. This has implications for the design of future experiments, and for the usefulness of Design Patterns that depend heavily on recursion in their structure.

Work task 2

*"After a code review, an incompetent manager requires you to change the method **operatorname()** to **varname()** on the **VarTerm** class only".*

This in effect broke the COMPOSITE pattern, because one of the concrete classes no longer followed the declaration of the superclass.

Hypotheses: We expected that the ALT group would find the pre-test easier than the PAT group (B4: $+$), because all the resulting changes only

have to be made in the class already being changed. The PAT group would have to modify the VISITOR classes. One of the modifications is somewhat tricky to spot, so we also expected lower correctness (B5: $-$).

In the post-test, we expected that the PAT and ALT groups would be roughly equal, because the magnitude of the change is the same in both cases (B6: 0).

Results: B4:$+ \triangleright +108\%^{+456}_{-22}$ seemed to be supported, in that the PAT group needed twice as long as the ALT group. However, we were still dealing with subjects who in five cases out of seven were not using VISITOR. There was quite a lot of confusion visible in the solutions, and the correctness was quite low B5:$- \triangleright -20\%^{+18}_{-40}$. Also, four subjects were missing (relative to work task 1), because they had run out of time.

B6:$0 \triangleright 18\%^{+132}_{-40}$ was contradicted; the correctness of the PAT solutions was markedly lower than that of the ALT solutions (average grade 2.8 vs. 3.8).

This reinforces the conclusion from work task 1: our subjects were badly confused by the VISITOR pattern; only 6 out of 21 achieved a "Correct" or "Almost correct" solution (PRE and POST combined).

1.5.3 Decorator: Communication Channels (CO)

This program is mostly a wrapper library. A communication channel establishes a connection for transparently transferring packets of data of arbitrary length. One can turn on additional functionality for logging, compression and encryption.

The library does not implement the functionality itself, but only provides a FAÇADE for a system library (whose internal source code was unavailable to the subjects). However, this application of the FAÇADE pattern is irrelevant to the experiment.

The PAT version uses a DECORATOR scheme to add extra functionality to a bare channel. It consists of six classes over 404 lines.

The ALT version has only a single class, using boolean flags and `if` sequences for turning functionality on and off during the processing of

one packet. It consists of 342 lines, and is the only instance where the ALT version has a structured (as opposed to object-oriented) design.

Work task 1

"Enhance the functionality of the program so that error correction can be added".

The actual encoder/decoder for the error correction was available as an interface with a working, hidden implementation. Its interface was exactly analogous to one of the functions that were already in use.

Hypotheses: DECORATOR has two competing influences. On the one hand, it nicely separates the functionality and all but removes dependencies, so that it should be easy to add a new function by adding a new DECORATOR, rather than having to find the right place in a block of `if` statements.

On the other hand, this separation also means that it is more difficult to trace what is actually going to happen at runtime, as opposed to a situation in which there is a structured block of code with a clear sequence of flag tests.

We expected the first influence to be stronger, and hence that it would be quicker to enhance the PAT version (C1: $-$), especially at higher levels of pattern knowledge (C2: $-$).

Results: C1:$- \triangleright +13\%^{+69}_{-25}$ was not supported, because the groups spent virtually the same time. However, the correctness of the solutions was much better for the PAT group, with a perfect (10/10) "Correct" score for all PRE-PAT subjects. In the PRE-ALT group, there were nine "Correct", one "Almost correct" and two "Right idea".

Expectation C2:$- \triangleright -49\%^{-28}_{-64}$ was strongly supported. Also the correctness was again better with nine "Correct", vs. six "Correct", two "Almost correct" and two "Right idea".

The conclusion here is that DECORATOR is a pattern that can be grasped with reasonable ease, and contributes to higher correctness. With training, its use also results in considerably faster development.

Work task 2

A communication channel—as implemented in the program—has an internal state (`open`, `closed`, `failed`) that is altered by certain operations. Work task 2 asked the subjects to *"determine when a **reset()** call will return the '**impossible**' result"*. This required the subjects to find where the underlying state was changed, and how. This should be easier for the ALT group, where the state changes are strongly localized, so we expected shorter time (C3a: $+$), (C3b: $+$) and higher correctness (C4a: $-$), (C4b: $-$).

The subjects were also asked to *"create a channel that performs compression and encryption"*. Again, the ALT group should have the advantage, since they only needed one `new` statement. The PAT group needed to determine the correct nesting of DECORATORs to achieve the same result.

Results: C3a:$+ \triangleright +117\%\,^{+260}_{+31}$ was strongly supported in the pre-test, but inconclusive in the post-test (C3b:$+ \triangleright +9\%\,^{+88}_{-37}$). Correctness was actually better for the PAT group in the pre-test, contrary to C4a:$- \triangleright +15\%\,^{+22}_{-5}$, and remained so in the post-test C4b:$- \triangleright -5\%\,^{+5}_{-10}$.

This seems to reinforce our conclusion that DECORATOR had a mainly positive effect and can be grasped without too much training. Any problems caused by the delocalisation that results from applying this pattern were outweighed by the greater ease of composition of functions.

1.5.4 Composite and Abstract Factory: Graphics Library (GR)

The Graphics Library enables creation, manipulation and drawing of simple graphical objects, such as points, lines and circles. They can be rendered to different displays (alphanumeric or pixel), represented as output device classes with standardized interfaces.

In a central class a device context (type) is selected, and depending on this choice different versions of the graphical objects are created. Some basic objects (points and lines) are implemented identically for all devices, but circles have special implementations per device. Objects can

also be collected in groups, which can then be manipulated like objects themselves.

The PAT version uses ABSTFACTORY for the generator classes, and COMPOSITE for the hierarchical object grouping.

The ALT version uses a single generator class with `switch` statements for the different devices, per object type. Combination and manipulation of objects is achieved with a quasi-COMPOSITE, the only difference being that there is no hierarchical group nesting.

This pair of programs has the smallest structural differences of all of the four pairs. The PAT version has 13 classes over 683 lines, the ALT version uses 11 classes over 667 lines. Both versions contain an identical main program that defines an Olympic logo with five circles and a line, rotates it 180° and draws it.

Output devices are represented by working classes with simple interfaces and hidden implementations, and can be run by the subjects. Some sample output is also included in the task documentation.

Work task 1

"Add a third device type (a pen plotter)". The PAT group had to introduce a new concrete factory class, extend the factory selector method, and add two concrete product classes using the supplied `Plotter` device interface. The ALT subjects instead had to extend the `switch` statements; the product classes were the same.

This task was also one in which there was a difference from the original experiment, because in the present case, subjects actually tried out their solutions and ran into issues with scaling, forgetting to select a visible pen, etc. These issues affect the "product classes", which are common to both the PAT and ALT versions. Since this extended the total time, we expected to see less difference overall than in the original experiment.

Hypotheses: The actual volume of changes is the same for the PAT and ALT groups, so the main difference should come from differences in comprehension. The ALT program, with its localized `switch` statements, should be easier to understand (G1: +).

Pattern knowledge should help both groups to deal with the COMPOSITE (G2a: $-$), while the PAT group may derive an additional advantage from better understanding of the ABSTFACTORY (G2b: $-$).

Results: G1:$+ \triangleright -17\%^{+40}_{-50}$ was inconclusive. If we ignore one ALT outlier, the groups were virtually identical with respect to time used.

However, the correctness was significantly better for the PAT group. Inspection of the code revealed that several of the PRE-ALT subjects did not use the supplied `Plotter` interface, but instead just copied one of the existing output device interfaces. However, this did not necessarily invalidate the test, since it is the structure more than the particular class that matters.

The post-test, G2a:$- \triangleright +2\%^{+76}_{-41}$, was inconclusive, with a hint in the opposite direction—subjects needed more time after the course for the ALT version. G2b:$- \triangleright +62\%^{+120}_{+19}$ was not supported: The PAT group needed more time than the ALT group; this time all the subjects actually implemented the plotter as intended.

It seems that use of ABSTFACTORY therefore had little influence on the time needed to make the changes, but it may have contributed positively to quality. This agrees with the purpose of the pattern: to concentrate the knowledge of which objects should be created in one place. The actual work remains roughly the same, but is localised instead of being spread out.

Work task 2

"Determine whether a given sequence of statements will result in an x-shaped figure". This is a comprehension test on COMPOSITE, where the key is to recognize that references, and not copies of objects, are stored in an object group.

Hypotheses: First of all, we expected the subjects to actually try running the function containing the statements (it is present in the program, but not called by default). This was in contrast to the original experiment, where analysis was the only possibility. Consequently, we expected correct answers.

The structure of both programs is similar, so no difference in time was expected for PAT and ALT (G3a: 0), (G3b: 0), but we did expect the post-test to be faster than the pre-test, due to knowledge about the COMPOSITE (G4a: $-$), (G4b: $-$).

Results: High correctness was present only for the POST-ALT group. All other groups had a significant number of incorrect answers, despite the fact that most of them tested their solutions (visible traces in the code logs).

G3a:0 $\triangleright$ $-9\%^{+86}_{-56}$ and G3b:0 $\triangleright$ $-39\%^{+32}_{-72}$ were inconclusive. The POST-PAT group spent 39% less time than the Post-ALT group, but their correctness was significantly lower. Perhaps they were tricked by the application of COMPOSITE and forgot to take the effect of object references vs. object copies into account.

G4a:$-$ $\triangleright$ $+66\%^{+242}_{-20}$ and G4b:$-$ $\triangleright$ $+11\%^{+139}_{-48}$ were not supported; there was no significant difference in time spent between the pre-test or post-test. If anything, G4a tended in the opposite direction, with subjects spending more time on the ALT version after the course. However, this may simply be due to fatigue.

Due to scaling, a part of the figure would have been clipped and invisible, so we conclude that a number of subjects tried the drawing method, and when they did not get a good visible result, they resorted to (faulty) analysis instead of getting a good test output. After that, we were probably measuring a combination of their C++ proficiency with pointers and their approach to testing, rather than knowledge of Design Patterns. However, the fact that they trusted their analysis rather than actually making the test run well is interesting in itself.

1.5.5 Summary of qualitative results

We found evidence for both the usefulness and potential for harm of using Design Patterns, mostly as predicted by software engineering common sense. In summary:

OBSERVER —expectation: The pattern solution is more complicated and harmful relative to the alternative solution, unless its flexibility is

required

Actual result: There was no significant harmful effect, even for subjects with little or no patterns knowledge. After a short course, a significant benefit was observed in terms of both time and correctness.

COMPOSITE, VISITOR —expectation: VISITOR is difficult to understand and thus harmful.

Actual result: Both before and after the course, the majority of the subjects *did not even use* VISITOR even though two examples were present in the code. The correctness of the solutions was significantly lower than in the alternate design.

DECORATOR —expectation: Delocalization of functionality is expected to make it easier to change, but more difficult to analyze and call.

Actual result: The first expectation was supported, but the second was contradicted. The correctness and time improved significantly after the course, but the correctness was also better in the PAT version before the course, for no large penalty in time spent.

COMPOSITE, ABSTFACTORY —expectation: The similarities in the designs of PAT and ALT versions lead us to expect only minor differences.

Actual result: No overriding difference was observed, even though the course helped with the ABSTFACTORY pattern, which was also present in the ALT version.

1.5.6 Other observed effects

In this Section, we discuss cases where a significant effect was observed, but no prior hypothesis existed. This situation arises because of the regression-based analysis model, which automatically estimates more coefficients than those needed to test the hypotheses of the original experiment. These results are necessarily of an exploratory nature.

Since there are no hypotheses to use for labelling, we will instead refer to the panel title in Figure 1.2 or 1.3, together with program abbreviation and task number. The observations are grouped by program and thus pattern.

Stock Ticker / OBSERVER

Figure 1.2 (time): *Course effects on design pattern programs, work task 2*—this panel compares the time spent on the work task before (pre-test) and after (post-test) the course, and there is a significant difference present. In the post-test, the subjects used 60% less time than in the pre-test.

In the corresponding coefficient for the ALT version, the opposite effect is present, though without being significant. So the effect of the course was to reduce the time used for the PAT version, and increase it for the ALT version.

Part of the explanation lies with two subjects; one spent some time looking for a typographical error (an extra closing brace that caused cryptic error messages from the compiler), and the other struggled with the syntax of arrays and enumerators. These two outliers increased the time spent by the ALT group; but being outliers, did not cause the increase to be statistically significant.

That the course should benefit the PAT group is as expected. The nature of the program and task were such that understanding of the OBSERVER pattern reduced the task to only a few lines. The corresponding panels in Figure 1.3 (correctness, course effects) show almost no discernible effects.

Figure 1.3 (correctness), *Effects of design patterns before course* and *Effects of design patterns after course* both point in the same direction: the solutions of the PAT version have a higher correctness. The effect is significant in the pre-test and close to significant in the post-test. It reinforces the conclusion in Section 1.5.5 and contradicts the expectation from the original experiment.

Communication Library / DECORATOR

Figure 1.2 (time): *Course effects on design pattern programs, work task 1*—the panel shows that the subjects who maintained the PAT version of the program spent significantly less time in the post-test than those who maintained it in the pre-test (50% less). In Figure 1.3, the panel

Effects of design patterns after course measures the difference between ALT and PAT version in the post-test, and shows a significant increase in correctness for the PAT version.

There is one outlier in the PRE-ALT group, who worked for about 90 minutes before submitting a large change for compilation. However, the measured effect persists even if this outlier is omitted. The tentative interpretation is that DECORATOR is a pattern that benefits significantly those who take even a short course in it, and that such benefit influences both the correctness of the solutions and the time used to complete them.

Base levels

The size and complexity of the programs varied, and this can be seen in the "base level" (time used for the ALT version in the pre-test, correctness at least 4 out of 5, as defined in Sections 1.3.6 and 1.3.7). Qualitative analysis of the solutions showed that recursion and recursive data structures created problems for a number of subjects. This affected the Boolean Formulas and Graphics Library programs (longer time, lower correctness), but is not a threat to validity because recursion is a central feature of the COMPOSITE pattern that is present in these programs.

The Stock Ticker program (OBSERVER pattern) was the shortest and simplest, and the base levels show a short time and high correctness for the solutions. Again, this reflects the pattern itself, which is structurally very simple.[4] In this experiment the subjects had few problems understanding and extending it, and the base level observations reinforce the conclusions regarding this pattern.

1.6 Comparison with the original experiment

This experiment was originally designed to investigate whether it is useful to utilize Design Patterns during program construction, even if

4. The subtle interaction effects which it is possible to encounter with multiple OBSERVERs and event-driven programming in general, become visible only in programs of greater complexity.

the particular problem can be solved in simpler ways. The context was program maintenance by developers other than the original ones, and the experiment used pen and paper.

Our replication of the original experiment added the dimension of a real programming environment, so we retained the original aim, while adding an interest in the effects of the differences in execution of the experiment. We therefore re-analyzed the data from the original experiment using the same regression model, estimation method and software as for our replication, thus making it easier to compare the two experiments.

The program Communication Library had three work tasks in the original experiment. The last two were quite similar and had similar hypotheses; in our replication they were combined to give a more symmetrical experimental design. When re-evaluating the data from the original experiment, the completion times for the last two tasks of this program were summed, and the correctness scores averaged and rounded to the nearest integer to match the analysis model. In the original experiment, so many subjects misunderstood Boolean Formulas Task Two that it was omitted from the analysis. We likewise omitted it from our re-evaluation.

1.6.1 Base level and variance

For the dependent variable *time*, the upper left panel of Figure 1.4 shows that the *trends* are similar, in that the same work tasks take a long or short time to complete. However, the absolute values differ. Those tasks that contained a lot of programming (Boolean Formulas task 1, Graphics Library task 1) took significantly longer in the replication.

Our explanation is that this is mainly an effect of the switch to actual programming; since all the details have to be correct, there is more work to be done than simply sketching out a class on paper. The variances are also larger, and we attribute that to the same effect—Prechelt has previously estimated the expected speed difference between fast and slow engineers to be on the order of 4:1 (**?**), and we expect both the

programming environment and the varied background of the subjects to contribute.

Another possible contributing factor is the fact that our subjects were paid for participating, whereas in the original experiment, the subjects were volunteers. We would expect this to cause a greater variance, because the volunteers would generally be more interested than a less self-selecting sample. However, the presence or strength of this factor is difficult to evaluate separately.

For the dependent variable *correctness*, seen in the upper left panel of Figure 1.5, the situation is similar, though the correctness of the solutions in our replication is often somewhat lower than in the original experiment. One possible explanation is that the criteria for scoring were not identical. For instance, it is possible to grade as "Correct" a paper solution that does not actually compile due to a syntax error; in our replication this situation would have given an "Almost correct" score at best.

1.6.2 Elapsed time

The four lower panels of Figure 1.4 show that in most cases, the sign, and to a lesser extent the size of the observed effects, are similar in both the original and the replicated experiments. Since the replication is fairly close, we concentrate the following discussion on the following cases: a hypothesis exists, its estimated actual coefficient changes sign and there is little or no overlap of the confidence limits. We first note that there seem to be no systematic differences. As the replication is fairly close, this is as expected, and improves our confidence in the validity of the experiment.

For Stock Ticker/OBSERVER work task 1, the hypothesis (S2: $-$) is contradicted in the original experiment and confirmed in the replication. The expectation was that given knowledge of Design Patterns, subjects would find it easier to implement the work task on the PAT version, because the programming effort would be much smaller. This did not happen in the original experiment, while the expectation was supported in our replication, leading to opposite conclusions regarding the

OBSERVER pattern.

In Communications Library/DECORATOR work task 1, there are conflicting expectations. The ALT version has a greater amount of localized code and should be easier to understand, but new functionality has to be added in several places and correctly sequenced, leading Prechelt *et al.* to expect the subjects working on the PAT version to be faster, even in the pre-test. In the original experiment, the first effect was the strongest, leading to a strong confirmation of (C1: $-$). In the replication, subjects working on the ALT version were marginally slower than those working on the PAT version, leading to an inconclusive result. Simultaneously, correctness was significantly higher for the PAT version than for the ALT version in the replication, and even more so in the original experiment. In summary, the subjects in the replicated experiment had more problems with DECORATOR before training than in the original experiment. In the post-test there is no significant difference between the two experiments.

1.6.3 Correctness

As with the dependent variable time, we see no large, systematic differences between the original experiment and our replication in the estimated coefficients for the dependent variable correctness. For Communications Library/DECORATOR work task 1, we observed a smaller improvement in correctness going from the ALT version to the PAT version of the program than in the original experiment. The explanation lies in the base level correctness (ALT version), which is higher for the replicated experiment, leaving less room for improvement.

Graphics Library/ABSTFACTORY recorded improvements in correctness between the pre-test and post-test for ALT version programs. We believe this to be caused by the fact that both the PAT and ALT versions of this program contained an ABSTFACTORY pattern as noted in Table 1.1, and that the subjects benefited from the course in understanding this pattern.

1.6.4 Summary

For OBSERVER, we did not find the negative effect that was observed in the original experiment, while for VISITOR we had a very strong negative indication. For DECORATOR also we found fewer harmful effects, while COMPOSITE / ABSTFACTORY turned out about the same.

The total and C++ experience of our subjects was roughly comparable to those of the original experiment, but our subjects had less pattern knowledge to start with. The materials, experimental design and Design Patterns course were the same. The main differences were a more heterogeneous group of subjects (multiple companies), and the programming environment vs. pen and paper.

Having a detailed log of all compilations enabled better understanding of what the subjects were doing while solving their tasks. In particular, several cases were observed in which a subject worked on one solution, discarded it and did something different, whereas in the final code, no trace of the intermediate solution was present.

1.6.5 Lessons learned

Our suggested "lessons learned" are similar to those in the original experiment. It is not always useful to use a pattern if a simpler design will do the job; and if the pattern is a complex one like VISITOR, even "proper" use can confuse more than it helps.

However, patterns that are more intuitive, such as OBSERVER or DECORATOR, will generally do little harm, and code based on them can be readily extended even by developers who do not fully understand them.

A developer needs an awareness, not just of Design Patterns or whatever other design method is in fashion, but also of good design principles in general. Good breadth in education and experience can make up for lack of knowledge of specific patterns, but the opposite does not follow. Educational institutions should avoid the temptation to concentrate too much on the current design fashion, whatever it might be.

1.7 Methodological results

During the course of this experiment we identified a number of issues relating to the conduct of such experiments, which we will summarize here:

1.7.1 Measurements

The subjects measured the total elapsed time for a task themselves, noting it in a questionnaire. They could also add free-form comments at the completion of every work task. Our logging system also measured the elapsed time, and unobtrusively saved time-stamped copies of every file compiled. After each task, all the subjects completed the questionnaire, grading the difficulty of the task, the helpfulness of any patterns present, and their own use of the patterns.

The combination of both machine- and self-measured elapsed time, together with the comments, enabled better verification of the actual measurement. The fact that there were no significant discrepancies increased our confidence that the times were measured correctly.

The free-form comments and the compilation logs were valuable in both grading the correctness of the solutions, and in later qualitative analysis.

1.7.2 Technical setup

Setting up a lab with 45-50 workstations is an expensive and laborious task. We instead decided to use a Terminal Server™ configuration, in which each subject would bring his/her own laptop computer (the vast majority of consultants now use them), connect to our network and then work entirely inside the Terminal Server environment. We used four servers with a total of six CPUs and 3GB of RAM. The servers ran at 20-30% CPU load during the experiment. Logs were made of various performance parameters to verify that no major server problems affected the experiment.

The subjects were assigned seating based on their group membership,

such that two subjects sitting next to each other never worked on the same task.

On the morning of the first day, the subjects were assisted by Simula Research Laboratory technical support staff to connect their laptop PCs to our network. This operation took from a few to about 15 minutes per subject, depending on their Windows configuration. There were three staff members on hand to handle this task, and this proved barely sufficient.

If the organization owning the laptop had a restrictive security policy, this relatively simple reconfiguration became very difficult or impossible to perform. We had 10 terminals ready to accommodate subjects who for any reason could not use their own computer, and seven of them were actually used. To avoid losing subjects it is essential to have such a backup option ready. Switching in the middle of the experiment should be avoided; in our case, if the laptop did not work satisfactorily on our network within a maximum of 15 minutes, the subject was given a terminal instead and the laptop was never used.

1.7.3 Programming environment

The environment was set up with a pre-installed editor/compiler, Web browser, a viewer for PDF documents, and nothing else. Access to other functions, files, etc., was removed or blocked. In this way we ensured that the subjects had equal working conditions, and also guarded against mishaps with incompatible header files and other setup-related problems. It should be noted that such problems are not easily overcome in the C++ world, where even "standard" headers are incompatible among major compiler manufacturers.

1.7.4 Big-bang experiments

This experiment had a design that forced all the subjects to be present at a certain place, and at the same time, for several days. Finding participants is quite difficult, because people have to be available at exactly the given time, which may be difficult to schedule for their employers.

In addition, if anything happens to prevent attendance, the subject is lost: there is no second chance.

An alternative model is to perform the experiment in batches, over an extended period of time. This is more flexible and robust, and also allows the experiment to be extended with more subjects if needed. However, it precludes experimental designs that require the subjects to have some common activity, such as the patterns course in this experiment.

1.7.5 Place of experiment

Due to the design of the experiment, our subjects had to come to Simula Research Laboratory, both for the course and to do the exercises. An alternative model is to use a web-based tool (**?**) to deliver the programming tasks, questionnaires and tools directly to the subjects, and have them perform their tasks in their own office environment.

The experimenters must in any case be in attendance on the premises to handle any problems.The major methodological challenge in this model is to keep control of the experiment; on the technical side it is not trivial to make sure that all the subjects' computers have equivalent and properly working environments. It is also more difficult to install and use various monitoring tools.

1.7.6 Recruitment and subject selection

The intended population for this experiment were programmers who make general-purpose data processing software. A number of consultancy companies were asked to participate in the experiment. They were told the general outline, consisting of a one-day patterns course between two half days of exercises. The subjects were paid for the exercise time, and received the course free.

Both the companies and individual subjects were told that they were participating in an experiment, but were not told about any of the goals, hypotheses or expectations, nor what was measured or how.

The participation was formalized by contracts signed by each company and Simula Research Laboratory. Each company was also allowed to

charge a limited number of management/overhead hours. Our experience is that allowing reasonable overhead costs increases the likelihood of participation.

The experimental design required all participants to be present at the same time, for two consecutive days. This made it much more difficult to get a sufficient number than for previous experiments that could be done on each company's premises, one company at a time.

1.7.7 Subject background mapping

The background data was collected using an on-line questionnaire prior to the experiment (**?**). The questionnaire used was identical to that used in the original experiment. However, instead of the subjective, manual process that was used to allocate subjects to group in the original experiment, the answers were transferred to a database, and processed by a scoring program. A score was calculated in each of the following three categories:

1. C++ programming experience and volume, measured in number of programming years and number of lines written;
2. Knowledge of design methods, measured by number of methods known and number of practical uses of each method;
3. Knowledge of patterns, measured by number of patterns known, and number of practical applications of each pattern.

Finally, the scores were combined with relative weights of $\frac{1}{6}$, $\frac{1}{6}$ and $\frac{2}{3}$, reflecting the relative priorities of the categories.

Using a scoring program made the relative weights of the different factors explicit and visible. However, no automated process can fully address all details of a subjects' qualifications, especially free-form comments that clarify the purely quantitative aspects. Manual inspection and sometimes adjustment is required, for example if a subject filled in "5" as the number of years of education, but stated that two of them were in high school.

1.7.8 Prequalification and blocking

Having too large a spread of education, experience and knowledge in the subjects can undermine the use of standard quantitative statistics, because it introduces individual differences that may mask the trait we seek to measure. As experienced in this experiment, it is difficult to predict the performance of subjects from indirect measures such as education or work experience, and balancing according to such criteria may therefore not be enough to ensure an actual balance with respect to the experimental tasks.

This can be mitigated by using nontrivial familiarization and calibration tasks that are common to all the subjects. It might also be necessary to exclude some subjects based on a pre-test, if it turns out that they are not qualified to take part. The sample is then no longer random, and great care must be taken to ensure that the subjects are (and remain!) a representative sample of the population that is being studied. We note that simply relying on participating organizations to select a sample (by deciding who to send) may not be enough.

One consequence is that the process of selecting subjects should be started several months in advance of the experiment, to allow time for sufficient iterations. "Big-bang" experiments are especially sensitive, because there is no second chance. As discussed in Section 1.4.3, this experiment was robust with respect to (lack of) group balancing. However, mapping the subjects' background is still important to help ensure a representative sample of the targeted population, and thereby improve the experiments' external validity.

1.7.9 Details matter

In pen and paper experiments, subjects do not have to write exactly correct syntax. Once a compiler is introduced this is no longer true, and technicalities that are viewed by the experimenter as irrelevant may consume significant amounts of time.

In the Boolean Formulas program, several subjects knew neither the truth table nor the C++ syntax for the `XOR` operator, and consequently

spent time on this.

In the Graphics Library program, one subject had never seen a pen plotter and consequently did not understand what to do. The experimenter therefore has to take extreme care to fully specify, document or avoid such details to improve data correctness.

The experimental design used here seeks to mitigate the consequences of such "details". All the three problems cited contributed to increase the variability of the data, or to data loss. By making multiple measurements on both PAT and ALT programs and tasks for each subject, the overall impact is minimised and the chance of systematic bias in the results lessened.

1.8 Threats to validity

An experiment is by definition an artificial situation. In this Section we address threats to both internal and external validity of the observed results. The discussion is in part based on the guidelines recently set forth by Kitchenham (**?**).

1.8.1 Threats to internal validity

Internal validity is the degree to which the observed effects depend only on the intended experimental variables. In this experiment, the main threats are from inter-individual, and inter-group, differences between subjects that mask the intended effects. The purely technical proficiency of the subjects (as distinct from their ability to understand program structures and patterns) is also a factor.

Group balancing

We have already described how the groups were balanced with respect to pattern knowledge, general programming experience and C++ experience. Recent experience with the programming environment (Microsoft Visual Studio 6.0) and Windows itself was checked and also found to be reasonably well distributed.

The regression model and estimation method chosen are robust with regard to group differences, since each subject is its own control: all the subjects perform the work tasks of all the programs (half ALT and half PAT versions).

As it turned out (see 1.4.3), the regression model revealed no significant effect of the pre-qualification score, so a fully random assignment would probably have been just as good. In an experimental design where each subject receives *either* treatment A or B, group balancing is much more important. However, balancing can never be relied on completely, and is susceptible to problems such as participants not turning up. In such cases, some kind of calibration task is needed to determine actual performance levels, and should be included in the statistical model (**?**).

Technical factors

The programs used for the tasks were originally designed with a minimal user interface that consisted of a declaration of a `Window` class with a few self-explanatory methods (`drawtext`, `drawline`, `erase`, `resize`). In the present experiment, that class was implemented so that the programs actually ran and could show a window; the well-known stream mechanism for creating console text output `cout << "Hello\n";` was also added.

The windows so constructed remained active even if the program was stopped in a debugger, so their output was visible. To avoid distractions, the implementation was carefully hidden. No knowledge of Windows, Microsoft Foundation Classes, Java or any other specific system was assumed or needed.

Logs were kept of server load parameters to determine if overloads or faults interfered somewhat with the experiment. Moreover, all compilations were inspected, and subjects were also encouraged to submit (as free text) comments about any technical obstacles they might have encountered.

C++ proficiency

All the subjects performed an initial, familiarization task in order to try out the programming environment and the user interface.

Individual differences in C++ experience and capabilities interfered with the time spent on the programming. C++ is a relatively complex language with fine syntactical and semantic distinctions. Developers who are not proficient can spend significant time on details.

To lessen this threat, all compilations were evaluated, and the editing time for compilations that were purely about syntactical problems was subtracted from the total time for each subtask. The procedure is described in Section 1.4.1 above. Comparisons of estimates of the regression model coefficients and their confidence intervals showed that the corrections added no systematic bias.

However, neither did they shrink the confidence intervals much. Corrections of this kind will always depend to some extent on the subjective judgement of those doing the grading. Also, since copies of the source files were made only when compilations were performed, any subjects who spent time on syntax *without* making compilations cannot be assigned corrections. For these reasons the corrections were not used in the final analysis.

Another factor, likewise determined by inspection of the solutions, is that several subjects implemented one solution to a large extent, only to abandon it and start again in a different way. This also increased the individual variations.

1.8.2 Threats to external validity

External validity is the degree to which the results can be generalized and transferred to other situations. Several differences between the experimental situation and real-world maintenance must be considered.

Maintenance by the original designers

In our experiment the maintainers were different from the original programmers, so the experiment is not applicable to maintenance by the original designers. Original designers would be expected to remember not only the actual design, but also much of the motivation for it.

The use of Design Patterns may not have much impact in that case; in the current context, we are not interested in improvements resulting simply from the fact that a design with patterns fits the problem better than some alternative design.

Design Pattern experience

Some maintainers may have more experience with Design Patterns than did our subjects. In that case, we would expect the beneficial effects of patterns to be greater. Thus, the experiment is conservative in estimating benefits of using Design Patterns.

This expectation is motivated by the significant improvements (with one exception) in maintenance speed after the patterns course. The exceptions can largely be explained by the course being too short, so that subjects attempted to use a pattern that they did not fully understand. Deeper knowledge will probably not make the situation worse!

Program size, task size and tools

Real-world programs are much larger than those used in the experiment. With a restricted time and money budget, this is a limitation that is difficult to overcome. In addition, real-world programs are sometimes less well documented, and changes may be larger and involve more than one pattern. The effects of such differences are difficult to predict on a general basis. There are undoubtedly interaction effects that can occur between patterns, but would not be visible in such an experiment.

Judging by the qualitative findings, we would expect the results to be generally transferable. Good or bad understanding of a structural pat-

tern implies similarly good or bad understanding of the structure of programs employing it, and any benefits or problems should therefore be applicable. As large changes are often made up of several small ones, program size will be a scaling factor, but will probably not alter the direction of the effect. Relatively small change tasks, of the same order of magnitude as those in the experiment, do actually occur in industrial settings (**?**).

Maintainers may be more familiar with the language, development tools, etc., than were some of our subjects. However, given the widespread use of consultants, and relatively high turnover of developers in general, the experiment may reflect the real world in this respect more accurately than one might desire.

Realism of Tasks

In the experiment, most work tasks consisted of adding features that corresponded in some way to features or functions already in the code. The subjects could therefore use parts of the existing code as templates for their solutions. In an industrial context this would not necessarily be true.

However, most software does not fundamentally change its nature during what is normally termed "maintenance". Most features added or modified during maintenance correspond to something already present; for instance, a new layout for an existing screen, the addition of a field, or the addition of another web page to an existing structure. We therefore do not consider this aspect of the experiment to be a significant threat, though it excludes "maintenance" such as porting a program to a new platform or rewriting it for a fundamentally different kind of database.

Domain Knowledge

Each program came from a different domain: Graphics, Formula manipulation, GUI/Presentation and Communication. The Design Patterns used (ABSTFACTORY, COMPOSITE, VISITOR, OBSERVER and DEC-

ORATOR) are not domain-specific. Lack of domain knowledge was therefore more a threat to internal than external validity.

Inspection of the code logs and comments made by the subjects revealed a few cases of domain unfamiliarity: one subject did not know what a pen plotter was, and several subjects had some problems with Boolean arithmetic (see Section 1.7.9). While this increased the uncertainty of the estimates of the relevant coefficients of the regression model, we do not consider it a threat to the external validity of the results concerning the *patterns* themselves.

Experimental stress

Finally, our subjects were working under artificial, experimental conditions. The closest analogy is an exam. Even if one were always sitting beside someone working on a totally different task, watching a relaxed neighbour while one struggles is quite stressful, on top of the knowledge that one is being measured in visible, and possibly invisible, ways.

Another difference is that it is possible to walk away from a nonworking solution, as actually happened with several subjects. In industry that option is simply unavailable. When it happens, projects tend to become highly visible failures.

Working under stressful and tight deadlines is not unusual in industry (**?**). The additional stress from the experimental situation is expected to add to the size of the individual differences, because some individuals handle the situation better than others. Given the design of the experiment, we do not expect the results to be systematically skewed by this, so they should still be transferable to an industrial context

1.9 Conclusions

We replicated the experiment performed by Prechelt *et al.* (**?**), which investigated the question whether it is useful (with respect to maintenance) to design programs using Design Patterns, even if the actual design problem is simpler than that solved by the pattern. Our replication

sought to increase experimental realism by using a real programming environment instead of pen and paper, and by using paid professionals from multiple consultancy companies as subjects.

Logging tools were used to collect copies of the evolving solutions while the subjects worked. Together with free-form comments made by the subjects, this formed the basis for a qualitative evaluation of the results. In addition, a regression-based approach using Generalized Estimating Equations was adopted for the quantitative statistics. This approach takes into account the correlations between multiple work tasks performed by each subject.

We found that each Design Pattern tested has its own nature, so that it is not valid to characterize Design Patterns as useful or harmful in general, at least in the context (maintenance by other programmers than the original developers) addressed here.

The OBSERVER and DECORATOR patterns were generally understood even by subjects with little or no previous patterns knowledge, and after a short course the value of the patterns, in terms of both development time and, to some extent, correctness of the solutions, increased. The COMPOSITE pattern, with its reliance on recursion, caused some problems. It may be that recursion is no longer in general use in this kind of software, and a possible cause is the availability of predefined container classes in most languages. The VISITOR pattern, which has a fairly complicated structure, extracted a high cost in development time and poor correctness. Many subjects actually ignored it even when presented with template solutions that used it (and were documented as such).

Our results differ somewhat from those found by Prechelt *et al.*, especially in the case of VISITOR and OBSERVER. While they found VISITOR to be without significant harmful effects, few of our subjects achieved a good solution with it, even after the course. By contrast, we observed no significant harm done by using the OBSERVER pattern.

Having not only the final solution, but also the intermediate steps (provided by the logging mechanism), made possible a more extensive quantitative analysis. Using a real programming environment was one

prerequisite for such logging. The realism was also increased by introducing the need to compile and test the solutions.

We also demonstrated that it is possible to perform experiments on this scale while using a realistic environment and tools, and professional, paid subjects. It is possible to use Windows Terminal Server to provide a pre-configured environment without having to set up each workstation individually. This can be combined with Web-based tools to deliver content and questionnaires to subjects, thereby enabling experiments with a larger number of subjects.

Paying subjects to participate, and allowing some overhead costs, make it possible to get professional developers as subjects. If the experimental design allows it, it is also possible for them to participate while being in their normal work environment. In future work, these factors can be combined to increase the realism of the experiments and address some of the traditional threats to external validity.

Only four Design Patterns were evaluated in the original experiment and this replication. One area for future work is to evaluate other Design Patterns in widespread use from a similar standpoint: what effect would their use have on future maintenance, for programmers with and without relevant Design Pattern knowledge. Another need is to evaluate Design Patterns in larger contexts. The programming tasks do not necessarily have to be much larger, but the software of which they are a part should be of a more realistic size.

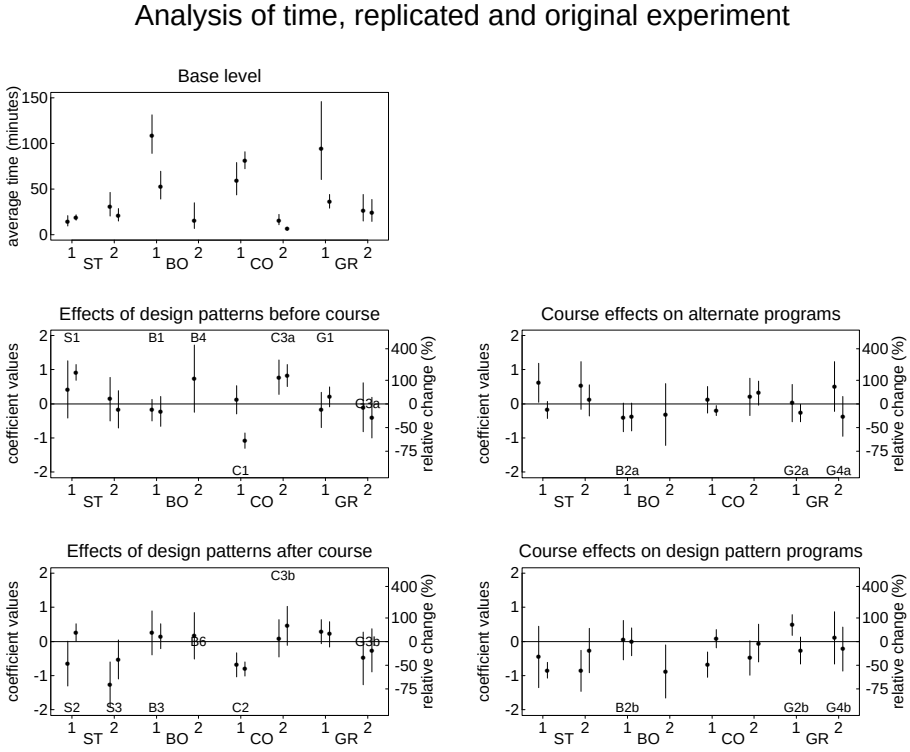

Figure 1.4: Completion times for all programs and tasks
Same format as Figure 1.2. For each coefficient, the left value is from the current replication, and the right value is from the original experiment, using the same analysis model and estimation method.

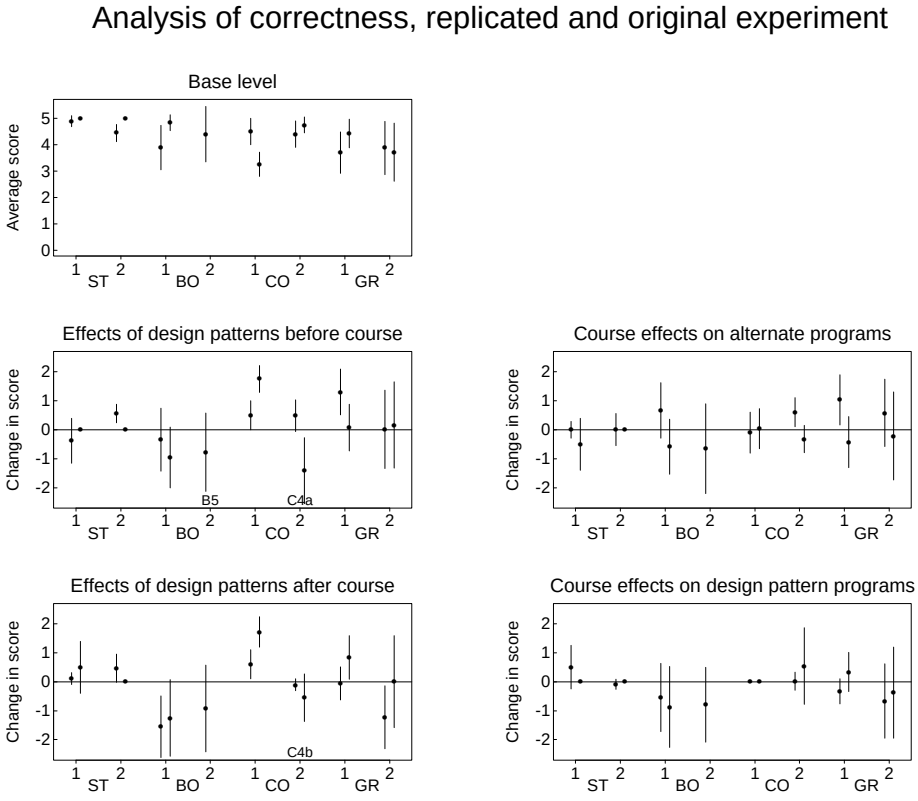

Figure 1.5: Correctness effects for all programs and tasks
Same format as Figure 1.3

Paper 2

Using a reference application with design patterns to produce industrial software

This paper was presented at PROFES 2004: 5th International Conference on Product Focused Software Process Improvement April 5–8, 2004 Keihanna-Plaza, Kansai Science City, Japan.

It appeared in the Proceedings: Bomarius, F., Iida, H. (Eds.), Product Focused Software Process Improvement. Vol. 3009 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, 2004.

Using a reference application with Design Patterns to produce Industrial Software

Marek Vokáč

Simula Research Laboratory, NO-1325 Lysaker, Norway

Oluf Jensen

EDB Business Consulting

Abstract

System architectures are described in abstract terms, often using Design Patterns. Actual reuse based on such descriptions requires that each development project derive a concrete architecture from the chosen Patterns, and then implement it in code.

This paper describes a case study of an industrial development project that adopted a *reference application* as a starting point, in order to avoid the need to design a complete architecture. Reference applications are usually made by platform or component vendors. Such applications are used to provide executable examples of best practices, for a particular platform, component set, or other technology. In this case study, the Pet Store application for the J2EE platform was chosen. Pet Store is documented by Design Patterns and is also available as source code. The development consisted of replacing the application logic, while keeping the architecture intact. The project was thus transformed from an *ab initio* development project into a major reuse/modification project. This development project was part of a software process improvement effort looking at processes *for* and *with* reuse.

Our results indicate that this approach works well, provided that the functional and non-functional requirements of the project match those of the reference application. The development time was shortened by approximately one third relative to the original estimate, and a well-designed application was produced despite lack of experience with the platform and n-layer architectures. Surprisingly, the production deployment to a different application server was still problematic, even though the original reference application was available as a sample for that server.

Key words: design patterns, reference application, Java Pet Store, industrial code, case study

2.1 Introduction

Software Design Patterns (Gamma *et al.*, 1995; Buschmann *et al.*, 1996; Rising, 1998) document solutions to well-known and defined problems, and thus provide an opportunity for architectural reuse. However, the structures and behaviour specified by a Pattern are at an abstract level and have to be translated into a concrete architecture, and then implemented in code, in each development project.

There exist other forms of architectural guidelines, such as reference architectures. These vary from very high-level and methodology-oriented approaches that describe organisational processes and general criteria for choosing between technologies, such as (Bernus and Nemes, 1996), to the more concrete, technological recommendations described in (Hallsteinsen and Swane, 2002), where possible components of a mobile application are described using prose and UML diagrams. Still, the burden of correctly designing and implementing the concrete architecture lies with the developers of each particular project.

By contrast, small pieces of sample code tend to illustrate a low-level solution to an isolated problem, and are more useful as examples than for direct reuse. While they often assume a particular architectural model, they do not explicitly specify or provide it.

This paper investigates a different approach: development by reuse and adaptation of a publicly available reference application. The starting point is a complete, working application that implements acknowledged best practices, with a concrete architecture documented by reference to well-known Design Patterns.

In our case study, the development was based on the Pet Store reference application available for the J2EE platform (Sun Microsystems, Inc, 2003). The development was performed by replacing the user interface, functionality and database structure, while leaving the architecture and application structure intact.

The decision to attempt large-scale reuse was partly motivated by the wish to quickly create a high quality Web application, in spite of a lack of experience with the J2EE platform and multi-layered architectures.

It was also motivated by an ongoing software process improvement project, in which the company cooperated with several research institutions. This project defined a refinement process investigating a development process for and with reuse.

This paper contains the following sections: Section 2.2 reviews reuse concepts and locates the present contribution in context. Section 2.3 presents the research methods, and Section 2.4 the development project that was studied. Results are described in Section 2.5, and Section 2.6 concludes and describes possible future work.

2.2 Background and concepts

A major challenge in developing a modern multi-layer, web-oriented application is to design an architecture that supports the required functionality on the specified platform, and to translate that architecture into running code. In this section, we look at some popular forms of software reuse, and define the salient concepts used in the present paper. These concepts form the basis for the present case study and the project that was investigated.

2.2.1 Forms of reuse

The tradition of software reuse dates back to 1968 (McIlroy, 1968) and earlier. Over time, several distinct modes of reuse have been described, as by Thomas *et al.* (1995). Their classification divided reuse into three kinds: *verbatim* reuse, where a component is not modified; reuse with *slight* modification; and reuse with *extensive* modification. In addition, we must take into account the kinds of artefact that are being reused.

At the architectural level, the artefacts available for reuse are abstract. Design Patterns (Sun Microsystems, Inc, 2002), books on Best Practices (Kassem, 2000; Alur *et al.*, 2001; Microsoft, Inc, 2003a), UML models (Hallsteinsen and Swane, 2002) and more general reference architec-

tures (Ciancarini *et al.*, 1998) must all be translated into concrete architectures suitable to the project at hand.

At a more concrete level, we find libraries and *frameworks* such as STRUTS (Apache Jakarta Project, 2003). Frameworks are used by plugging new components into them, to leverage existing functions, and to a certain extent, they dictate or encourage a particular architectural direction.

A concept that has recently received some attention is the *product family*, as described in van der Linden and Muller (1995); van der Linden (2002). When individual applications cover overlapping sets of requirements, there is a potential for a product family with shared modules. This often results in frameworks, libraries or other constructs meant to foster reuse between the applications.

In his seminal paper, Brooks (1987) argued that the *essence* of software lies in its complexity, conformity, changeability and invisibility; most of the improvements in the field of programming have addressed *accidental* factors, such as high-level languages and system response times (and thereby development cycle speeds). In the same way, plugging components into a framework or reusing sample code snippets may not solve the difficult problem of designing an architecture that conforms to a complex combination of functional, non-functional and platform requirements, while providing reasonable flexibility for future development. The task becomes especially difficult when time constraints tempt the project developers to formulate simple solutions to immediate problems.

2.2.2 Reference applications

Our work is based on the concept of a *reference application*. Such applications are made available by platform (Sun Microsystems, Inc, 2003; Microsoft, Inc, 2003b,c) or component (Rational, Inc, 2003; Infragistics, 2003) vendors for public use. They share a number of defining features: they are complete, running applications; full source code can be downloaded and dissected; they implement current best practices in the area they are meant to illustrate; and they are very well documented.

We are particularly interested in the fact that a reference application is executable, and provides full source code in addition to architectural descriptions. In doing so, it bridges the gap between abstract prescriptions for ”good” architecture, and the mass of details that must be considered in any final implementation. This is one of the more difficult parts of the design process, and a reference application provides a fully implemented answer-within the requirements set by its authors.

By basing the development on Sun’s Pet Store reference application, the project became somewhat similar to a product family project. However, the ”original” application was not an in-house product but instead a publicly available reference application. The development work belongs to the ”reuse with extensive modification” kind at the code level, but ”reuse with slight modification” at the application and architectural levels.

2.2.3 Patterns versus code

The relationship between the abstract and fairly simple description provided by a Design Pattern and the complex structure that may arise from its implementation can be illustrated by looking at two central patterns in the Pet Store reference application.

One of the most central Patterns is the Model-View-Controller (Sun Microsystems, Inc, 2002), which seemingly consists of three components. However, as implemented in Pet Store, the Model part consists of Enterprise JavaBean objects that access the database and process data, the role of Controller is performed by a set of objects that handle incoming requests, and the View is generated from screen templates combined with dynamic data.

In this way, what is conceptually a fairly simple pattern gives rise to a large number of objects (4+ for Model, 7 for View, 7 for Controller) for a single Use Case. The architecture provides considerable flexibility for screen and interaction design, and separates the business and data access logic from the presentation. However, correctly deriving and implementing such a design ab initio is not trivial.

At the opposite end from the presentation layer, we find the database. This component is often specified by the customer, because choosing the company DBMS tends to be a strategic, high-level decision.

There are several versions of the SQL standard, and each vendor has specific additions, extensions and limitations. It is therefore often necessary to make changes to applications when switching from one database to another. For instance, the syntax and rules for OUTER JOIN varies, and choosing the correct cursor type may have critical impact on performance. If SQL statements are spread throughout the code, this becomes a more difficult task.

The Data Access Object pattern specifies that one should make an object that encapsulates access to a particular resource, while hiding the actual implementation of the storage mechanism. This is a way of partially achieving "persistence transparency" as defined in ISO (1998). In the general case, this can be a large task, but implementing access objects for particular classes is more tedious than difficult.

2.3 Research methods

Our research design is a single case study with one study unit—the development project (Yin, 2003). The study was initiated after the development project was mostly completed, and was therefore conducted retrospectively. The research was performed as part of the Software Process Improvement through Knowledge Engineering (SPIKE) project, a research/industry collaboration partly funded by the Norwegian research Council.

Data was collected from project documents and logs, including the source code and UML designs. A full-day post-mortem seminar where all the developers participated was conducted. In addition, individual developers and representatives of the customer were contacted as needed to collect and verify information.

One of the authors (O. J.) served as development project manager and contributed first-hand experience.

2.4 The studied Development project

The Norwegian public sector actively attempts to provide 24-hour service for its citizens by using the Internet. The development of publicly accessible portals makes it possible for anyone to obtain certain services, which would otherwise only be available during normal working hours, at any time. Examples include Inland Revenue services (including tax returns), work and employment services, Social Benefit services and Municipal services.

The purpose of this project was to develop one such service, which makes it possible for the public to apply for a driver's license over the Internet, and which implements a secure and automated application process:

- The information received from the applicant is checked for correctness " The application is checked against the rules and regulations governing driver's license applications.
- Once the application for a driver's license is accepted, it becomes available to a public servant employed by the Road Authority, who is then responsible for processing the application.
- The applicant is automatically given a coded reference number by e-mail. This makes it possible for her to check the status of her application at any time.
- Once the application has been processed and accepted, the applicant may then use her reference number to select a date for her driver's license test.

The inception phase of the project produced a Requirements report containing specifications on several levels: Background and Purpose of the application; Terminology: methods and tools; a Business Process Model description; a Use Case Model description; a High level Business Class skeleton; sketches of User Dialog and Forms (HTML), and Database Models. This report also ensured that the customer's basic requirements were well documented. A short description of two central Use Cases follows:

USE CASE Registration of license application

The following information is registered:

- Personal information: First name, Last name, social security number, home address and e-mail address
- Applying for: First time applicant, additional vehicle classes
- What vehicle class
- Health information

The submitted information is checked against public registers and applicable regulations before it is accepted and stored as a driver's license application.

USE CASE Application status

Any applicant can look up her own application to view its status. The applicant must use the reference number she received via e-mail when the application was accepted.

2.4.1 The Pet Store reference application

The Pet Store reference application is presented in the book "Designing Enterprise Applications with the J2EE Platform" by Singh *et al.* (2002). It conforms to our general definition of a reference application:

> Its goal has been to introduce enterprise developers to the concepts and technology used in designing applications for the J2EE platform, and to give practical examples of a typical enterprise application.

Furthermore, Pet Store is an example of a particular kind of J2EE application:

> The Pet Store application is a typical e-commerce site. The customer selects items from a catalogue, places them in a shopping cart, and, when ready, purchases the shopping cart contents. Prior to a purchase, the sample application displays the order: the selected items, quantity and price

for each item, and the total cost. The customer can revise or update the order. To complete the purchase, the customer provides a shipping address and a credit card number.

Pet Store implements a number of hierarchically ordered Design Patterns. For instance, the MVC architectural pattern is implemented using Front Controller and Composite View, and database access in the Model part is performed by objects conforming to the Data Access Object pattern. This illustrates which tiers the application uses, and how to distribute the functionality across the tiers.

Sun's documentation (Sun Microsystems, Inc, 2003) consists of a high-level architectural overview, followed by a description of the relevant patterns, with references to actual classes and objects in the Pet Store application.

This provides both a pattern-based description and a working implementation with all the details filled in. The application is executable (provided one sets up a database server), and can be traced/debugged to find the actual order of execution.

The Pet Store reference application has been the subject of some debate in various Web forums, especially following its use in a benchmark to compare the J2EE and .NET platforms (Almaer, 2002; Öberg, 2003; Ditzel, 2003). It is also referred to in articles such as (Reimer and Srinivasan, 2003), on analyzing the usage of exceptions in large Java systems, and it is used in university curricula, e.g. (Ngu, 2003). It is therefore relatively well known among Java/J2EE practitioners.

2.4.2 Development methods

In addition to simply developing a new application for a customer, the company also wished to improve its software development process. To achieve this, the software process improvement project was established in collaboration with research institutions. One of the goals was to find and establish a refinement process, processes and methods *for* and *with* reuse.

The implementation of processes *for* reuse implies that any future

development project would have access to a "toolbox" of reusable elements, such as model elements, components or design patterns. The implementation of processes *with* reuse implies that development projects have knowledge of, and make use of this toolbox of reusable elements. In an example of successful reuse at Matra Cap Systemes (Henry and Faller, 1995), the process was jump-started by focusing on projects that could produce short-term gains while at the same time laying the foundation for more long-term reuse. The same approach was adopted here, by using the Driver's License project as the starting point for an improvement process. A crucial process was that of choosing a reuse strategy for the development project.

Key factors in the evaluation of reuse strategies were that the developers had relatively little experience with the J2EE platform and multi-level architectures. They therefore wished to find a form of reuse that did not require the design and full implementation of a complex architecture from abstract descriptions.

Given these considerations, and with reference to the various forms of reuse outlined in Section 2.2.1, the project chose to look closely at Sun's Pet Store reference application. This revealed that Pet Store's functionality had a sufficient level of similarity to the new application, as detailed in the next two sections.

2.4.3 Functional requirements matching

The refinement process suggests looking for matching functional requirements. To the development team, the functional similarities between the two applications were clear, as illustrated in the following table:

The database structure used in Pet Store did not match the legacy database that the Driver's License application was to use. It was expected that the Data Access Object pattern used in Pet Store would shield the rest of the application from the necessary modifications.

Driver's License	Pet Store
The applicant selects the service and type of license	The customer shops by selecting items from a catalogue
The applicant applies for a license	The customer places an order
The applicant fills in personal details	The customer fills in personal details
Personal details are verified against public registers; rules and regulations are checked against a rule database	Credit card data are verified externally; order consistency and completeness are verified by ERP or other systems
The applicant receives a confirmation e-mail with a case ID	The customer receives a confirmation e-mail with an order ID
Other (legacy) systems may process the same information	An administration manager reviews stock and enterprise financial data

Table 2.1: Functional mapping between the Pet Store and Driver's License applications

2.4.4 Non-functional requirements matching

Simply matching functional requirements is not sufficient to ensure successful reuse; non-functional requirements also have to be considered. Figure 1 shows a logical architecture that fits both the Driver's license and Pet Store applications:

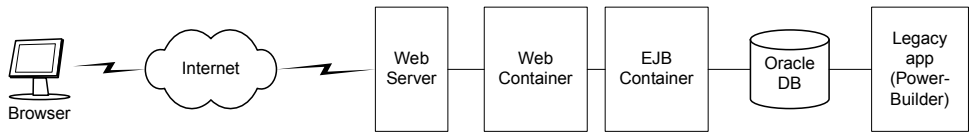

Figure 2.1: Logical architecture of the Driver's License application

The product had to satisfy a number of environmental and other non-functional constraints. Pet Store matched these requirements quite well, with some exceptions:

Driver's License	Pet Store
Platform: J2EE v. 1.2	J2EE v. 1.2
3- or n-layer architecture	3-layer architecture
Browser: IE5 and equivalent	Web pages are generated from tagged templates, adaptable to different browsers
Data storage in legacy database	Data is stored through Data Access Objects, adaptable to requirements; *possible mismatch*
Deployment platform uncertain	Pet Store is a reference/sample application included by all major platform vendors
Project size: $\leq$ 10 forms, $\leq$ 20 database tables	Pet Store is a relatively simple e-commerce application of comparable size
Rigorous validation of personal details needed	Incoming data (orders, addresses) assumed to be valid; *mismatch*

Table 2.2: Non-functional mapping between the Pet Store and Driver's License applications

2.4.5 Application server compatibility

The Pet Store application has been used as a benchmark for testing application server conformance. For instance, IBM includes it on the IBM WebSphere Application Server V4.0 for Windows NT product CD. Similarly, Oracle provides a Pet Store implementation for their Oracle9iAS Containers for J2EE (Oracle, Inc, 2003).

The Pet Store application is billed as a reference for both architecture and implementation by Sun, the vendor most involved in defining and implementing the J2EE platform. It seemed reasonable to have a high degree of confidence in the compatibility of the application with all the major server platforms.

It was therefore expected that basing the Driver's License application on Pet Store would make it simple to port it to different application

servers. This was an important requirement, since the customer for the system had not made the final choice of application server at the time the project began. A server change was consequently almost unavoidable.

2.4.6 Other factors

Basing development on Pet Store was expected to give the development team a number of advantages not available with other forms of reuse:

Development and implementation would be carried out by replacing all modules within each tier with project-specific code. The Pet Store architecture would be retained while replacing the actual functionality, avoiding the need for an architectural design, prototyping, etc.

Because the application would be executable from day one, regressions were expected to be easy to identify. The developers, project management and the customer would be able to see a running version that implemented the full production architecture at frequent intervals.

Finally, the total time used in development was expected to be less than for a normal development process, since development effort could focus directly on implementation, bypassing architecture and design. The project management expected to be able to show a working subset of the application to the customer after only two iterations, at least one iteration less than would otherwise have been the case. An iteration normally lasted about 3 weeks.

2.5 Results

Our case study found results that can be related to four different areas of the development project: organization; implementation; data structure and security; and deployment.

2.5.1 Project organization

At the outset, the project team decided on a software development process that resembled the Unified Process approach (Larman, 2001). The U.P. promotes iterative development, and describes iterations within the following phases: Inception, Elaboration, Construction, and Transition.

With a working application as a basis, iterations could be run in parallel in two phases that are usually performed in sequence: the Elaboration phase and the Construction phase. The Elaboration phase could focus more on identifying high-risk functionality and non-functional requirements, while the Construction phase focused on the implementation of a stable executable subset of the final product. At the same time, a Design and Implementation model was produced through reverse engineering at regular intervals. Otherwise, design and architectural activities were largely considered *unnecessary*, since the development effort was "reuse with only slight modifications" at the architectural level.

On average, six developers were involved. The development team was initially organized according to the defined tiers: two people were responsible for the client tier, two were working on the EJB tier, one person was working on the WEB tier, and one person was responsible for the database tier. Earlier experience by Frederick (2003) suggests that this approach has a number of benefits.

However, as development work progressed, it turned out that it would be better to reorganize work according to what design patterns a team member was working on. In other words, when a person was working on the Front controller pattern (which starts in the WEB tier), then he would also be responsible for the controller-related objects for that function in the EJB tier.

This provided team members with cleaner interfaces between each other's functionality and design patterns, and clear functional responsibilities. The project management also had a better control of responsibility for separate functionality. A consequence was better management and implementation of change orders from the customer.

However, there were exceptions to this rule. The people working on the client tier were, in the early stages, very occupied with the design of the application forms, and parameterization to the WEB tier.

2.5.2 Implementation

The implementation discipline had to produce a working subset of the final product at regular intervals, maintain the implementation model, and solve data validation and security issues. At the outset, the project had a working Pet Store enterprise application. This was now to be transformed into a Driver's License application enterprise system.

The prime use case was defined as Registration of license application (Section 2.4). The first functional subset to be produced had to receive and process a driver's licence application, through all tiers, and store it successfully in the database. This implied connecting all the different tiers so that application form data could be properly processed and stored in the database.

Connecting a new set of client tier web pages to the WEB tier proved simple. Once the web pages were designed, it was largely a parameterization task to secure a connection to the various handlers on the WEB tier. This is largely due to the screen definitions held in xml files, inherent in the Pet Store design.

Similarly, the connection between the WEB tier and the EJB tier proved to be simple, as the event handling mechanism is largely independent of the events it handles.

2.5.3 Database structure and data security

The data validation and security issues clearly belonged to the database tier. Solving these issues in this tier caused no conflicts with the functional responsibilities of the rest of the development team.

The structure of the legacy database was completely different from that of the Pet Store database. This required changes in the bean-managed persistent entity beans, and their Data Access Object pattern. The inter-

action between the Controller administrating calls to the database, and the entity beans performing the calls, also had to be reworked.

The validation and security issue was caused by a fundamental mismatch between an assumption underlying Pet Store and the Driver's License application. Since the Pet Store is a shop, it assumes that its customers will provide valid data and simply accepts the data proffered. However, in our case there are extensive rules and regulations to take into account. Also, since the end result-a driver's license-is a valuable legal document, there is a danger of malicious misuse and fraud. It is therefore necessary to validate and protect the data in what was previously a closed, in-house system at the Road Authority.

The security issue was resolved by storing driver's license application information in temporary tables until the system approved the application. Triggers and stored procedures ensured that accepted and approved application data was moved to the proper tables. This process was not part of the Java application logic at all. Application data that was not accepted was discarded. Both the database structure and security issues required several iterations before they were satisfactorily closed.

In the first iteration, no such temporary tables were implemented. In addition, only page one of the application form was stored in the database. Then, step by step during the following iterations, data from all application form pages were stored securely in the database. This had no negative impact on the progress of the rest of the project.

2.5.4 Deployment

Deployment on the production platform was the focus of the transition phase. During the preceding elaboration and construction phases, all implementation and deployment were performed using the Oracle OC4J application server. This application server is very easy to use, enabling the development team to produce subsets of the final product at short intervals.

Six months into the construction phase, the customer decided to im-

plement its new public portal on the IBM Websphere 4.01 application server. The transition to this new application server proved to be difficult. Both the original Pet Store application and the Driver's License application failed to deploy to this server.

This came as a surprise to the development team. If the Pet Store enterprise application acts as a reference application, then it should be able to run on all application servers that are certified to run J2EE. Conclusions derived from researching the issue can be summarized as follows:

- Different application servers have different deployment descriptor files, and these are organized differently.
- OC4J had a simpler and less restrictive RMI check, while Websphere required a strong RMI check.
- The classloaders of Websphere caused problems, and may be related to the strong RMI check.
- The Websphere single server edition behaved differently than the advanced edition.
- Websphere generally proved very difficult to work with (Aeinehchi, 2002). The compile, deploy, and run cycle was slow and required a lot of machine resources. Debugging was difficult, and often gave little valuable information.

2.6 Conclusions and future work

Our case study investigated a commercial project to develop a Web-based public service, for applying for driver's licenses. Having completed the requirements documentation, the project had a choice between several possible strategies for the actual development. One possibility was to design and implement the application from the ground up, using available tools in the form of Design Patterns, documented Best Practices and sample code.

Alternatively, it could be designed around a commercial or public domain framework or library. Ultimately, the team chose a different approach, by basing development on a reference application.

The developers had relatively little experience with the J2EE platform and multi-level architectures. They therefore wished to find a form of reuse that did not require the design and full implementation of a complex architecture from abstract descriptions, such as Design Patterns or architecture guidelines. The Pet Store application had a sufficient level of functional and non-functional similarity to the new application that it was chosen as the basis for the new application.

The development team, together with central members from the participating research institutions, performed a post mortem analysis on the project experience. The following sections summarize the results.

2.6.1 Positive experiences

Through Pet Store, the project possessed a basic architecture that employed the current best practices on J2EE design patterns and technology. Since the application was already implemented, little effort needed to be focused on architectural issues, and most effort was expended on implementation activities.

The learning curve for the team members was very good. This was definitely a problem-based learning approach, while at the same time the textbook answer was readily at hand.

The development tasks were considered professionally challenging, both with regard to functions of the final product, and to the implementation in a new and immature technology. The team motivation was therefore high, even though this was no longer a ”pure” development project.

Collaboration within the group was very good. Because of the chosen approach, there was always an executable application, and it was therefore easier to coordinate the efforts of the team members. The problem of people working for an extended period on an isolated problem and thereby drifting away from the group was avoided.

The total time and effort spent on the development matched expectations quite well. There is no doubt that this approach worked well for the application, and resulted in a combination of flexible architec-

ture and short time to market that would otherwise have been hard to achieve.

2.6.2 Negative experiences

Deployment on a production application server was expected to be easy, since Pet Store-the reference application used as a basis-is supplied as a sample application by the platform vendors. However, this turned out not to be the case, and even with assistance from the vendor, it was difficult to deploy both Pet Store and the new application on the production server.

The database design in Pet Store is stand-alone, and the application architecture does not contain provisions for interfaces and adaptation to existing database schemas (as opposed to purely technical adaptation to DBMS servers, which is covered by the Data Access Object pattern used). The database structure issue was solved by rewriting most of the data access objects, so here the benefit of using Pet Store as a basis was small.

The mismatch between the assumption of user trustworthiness between Pet Store and the Driver's license application is an example of a non-functional requirement that has architectural implications. When evaluating a development approach, it is important to check both functional and non-functional requirements closely. Special attention should be paid to non-functional requirements, even if it can be more difficult than checking functional requirements.

2.6.3 Conclusions

Basing new development on the code of an existing, well-documented application can be a viable method of reuse. Provided that functional and non-functional requirements match, the actual domain of the reference application and the new application need not be the same. Database structure and security are two areas where particular care must be taken, and where the potential for reuse may be limited. The Pet Store application by Sun is sufficiently well structured and docu-

mented to be usable in this role.

2.6.4 Future work

In most e-commerce applications, there is a front end that presents what products are available, and accepts user information. Other tiers are responsible for processing and storing this information, and producing a response that is then presented to the user by the front end. Data is stored in a database and passed on to other systems for further processing.

In the Pet Store reference application, this functionality is described at two different levels: at an analytical and design level, through a set of design patterns, and at an implementation level through the implementation of Pet Store.

The software improvement project defined a refinement process that would produce a toolbox containing an increasing number of refined model elements, frameworks, components and design patterns.

The elaboration phase of the development processes should also contain a refinement discipline, a discipline *with* reuse. The heart of this discipline is a search for similarities at many levels: functional and non-functional requirements, architecture and code.

Conversely, refinement *for* reuse should aim at producing artefacts at many levels. One way would be to attempt to refine a Pet Store type application into a set of design patterns (a pattern language for e-commerce applications?), and their equivalent components.

Then, when a similar e-commerce application is to be developed, the development process would mostly be a process of connecting a set of well-defined design patterns and their components into a working application.

Some of the preconditions needed to succeed using the reference-application approach were stated in the sections on functional and non-functional requirement matching (2.4.3–2.4.6). Further work is needed to determine a more general specification of the necessary and sufficient conditions.

Acknowledgments

The authors are most grateful to Prof. Dag Sjøberg, Prof. Ray Welland and Andrew McDonald for their extensive input and comments. This project was in part funded by the Norwegian Research Council as part of the SPIKE project. We would also like to mention the entire project development team at EDB Business Consulting, without whom this project would not have been a success.

Bibliography for paper 2

Aeinehchi, N., 2002. **Do NOT Use WebSphere Unless You are BLUE**. URL `http://www.theserverside.com/reviews/thread.jsp?thread_id=13639`

Almaer, D., 2002. **Making a Real World PetStore, TSS Newsletter #31**. URL `http://www.theserverside.com/resources/article.jsp?l=PetStore`

Alur, D., Crupi, J., Malks, D., 2001. **Core J2EE Patterns**. Prentice-Hall, Upper Saddle River, NJ, USA, ISBN: 0130648841.

Apache Jakarta Project, 2003. **STRUTS Home Page**. URL `http://jakarta.apache.org/struts/`

Bernus, P., Nemes, L., 1996. **A Framework to Define a Generic Enterprise Reference Architecture and Methodology**. Computer Integrated Manufacturing Systems 9 (3), 179–191.

Brooks, F. P. J., 1987. **No Silver Bullet: Essence and Accidents of Software Engineering**. IEEE Computer 20 (4), 10–19.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. **Pattern-Oriented Software Architecture**. Wiley, Chichester, ISBN: 0 471 95869 7.

Ciancarini, P., Tolksdorf, R., Vitali, F., Rossi, D., Knoche, A., 1998. **Coordinating Multiagent Applications on the WWW: A Reference Architecture**. IEEE Transactions on Software Engineering 24 (5), 362–375.

Ditzel, C., 2003. **Charles's Corner: Java Technology Pointers**. URL `http://java.sun.com/jugs/pointers.html`

Frederick, C., 2003. **Extreme Programming: Growing a Team Horizontally**. In: Marchesi, M., Succi, G. (Eds.), **XP/Agile Universe 2003**. Vol. 2573 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, pp. 9–17.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Boston, MA, USA, ISBN: 0201633612.

Hallsteinsen, S., Swane, E., 2002. **Handling the Diversity of Networked Devices by Means of a Product Family Approach**. In: **Software Product-Family Engineering**. Vol. 2290 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, pp. 264–281.

Henry, E., Faller, B., 1995. **Large-Scale Industrial Reuse to Reduce Cost and Cycle Time**. IEEE Software 12 (5), 47–53.

Infragistics, 2003. **Expense Application—Reference Application**. URL `http://www.infragistics.com/products/thinreference.asp`

ISO, 1998. **ISO/IEC 10746: Information Technology—Open Distributed Processing – Reference Model**. URL http://www.iso.org/iso/en/CombinedQueryResult.CombinedQueryResult?queryString=10746

Kassem, N., 2000. **Designing Enterprise Applications with the J2EE Platform**. Addison-Wesley, Boston, MA, USA, ISBN: 0201702770.

Larman, C., 2001. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process**, 2nd Edition. Prentice Hall, Upper Saddle river, NJ, USA, ISBN: 0130925691.

McIlroy, D., 1968. **Mass Produced Software Components**. In: Naur, P., Randell, B., Buxton, J. (Eds.), **Software Engineering: Concepts and Techniques. NATO Conferences**. Petrocelli, Garmisch, Germany, pp. 138–156.

Microsoft, Inc, 2003a. **Application Architecture for .NET: Designing Applications & Services**. Microsoft Press, Redmond, WA, USA, ISBN: 0735618372.

Microsoft, Inc, 2003b. **Duwamish 7.0**. URL http://msdn.microsoft.com/netframework/downloads/samples/?pull=/library/en-us/dnbda/html/bdasampduwam7.asp

Microsoft, Inc, 2003c. **Microsoft .NET Pet Shop 2.0**. URL http://msdn.microsoft.com/netframework/downloads/samples/?pull=/library/en-us/dnbda/html/bdasamppet.asp

Ngu, A., 2003. **CS5369A Enterprise Application Integration**. URL http://www.cs.swt.edu/ hn12/teaching/cs5369/2003Spring/admin/intro.html

Öberg, R., 2003. **Review of "The Petstore Revisited: J2EE vs .NET Application Server Performance Benchmark"**. URL http://www.google.com/search?q=cache:8OPCFEFDFd0J:www.dreambean.com/petstore.html+petstore+java+experience&hl=en&ie=UTF-8

Oracle, Inc, 2003. **Oracle9iAS Containers for J2EE User's Guide Release 2 (9.0.2)**. URL http://otn.oracle.com/tech/java/oc4j/doc_library/902/A95880_01/html/toc.htm

Rational, Inc, 2003. **PearlCircle Online Auction for J2EE**. URL http://www.rational.com/rda/wn_2002.jsp?SMSESSION=NO#pearlcircle

Reimer, D., Srinivasan, H., 2003. **Analyzing Exception Usage in Large Java Applications**. In: Romanovsky, A., Dony, C., Knudsen, J. L., Tripathi, A. (Eds.), **ECOOP '03: Workshop: Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms**. Darmstadt, Germany, pp. 10–19.

Rising, L., 1998. **The Patterns Handbook**. Cambridge University Press, Cam-

bridge, United Kingdom, ISBN: 0521648181.

Singh, I., Stearns, B., Johnson, M., Team, E., 2002. **Designing Enterprise Applications with the J2EE Platform**. Addison-Wesley, Boston, MA, USA, ISBN: 0201787903.

Sun Microsystems, Inc, 2002. **J2EE Patterns Catalog**. URL `http://java.sun.com/blueprints/patterns/j2ee_patterns/index.html`

Sun Microsystems, Inc, 2003. **Java Pet Store Demo 1.1.2**. URL `http://java.sun.com/blueprints/code/jps11/docs/index.html`

Thomas, W. T., Delis, A., Basili, V. R., 1995. **An Analysis of Errors in a Reuse-Oriented Development Environment**. Tech. Rep. CS-TR-3424, University of Maryland, Institute of Advanced Computer Studies, USA.

van der Linden, F., 2002. **Software Product Families in Europe: The Esaps & Café Projects**. IEEE Software 19 (4), 41–49.

van der Linden, F., Muller, J., 1995. **Composing Product Families from Reusable Components**. In: **1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, 1995**. IEEE Computer Society, Tucson, AZ, USA, pp. 35–40.

Yin, R., 2003. **Case Study Research, Design and Methods**, 3rd Edition. Sage Publications, Thousand Oaks, CA, USA, ISBN: 0-7619-2552-X.

Paper 3

Defect frequency and design patterns: an empirical study of industrial code

This paper appeared in IEEE Transactions on Software Engineering, vol. 30, issue 12, pp. 904–917, Dec. 2004.

Defect Frequency and Design Patterns: An Empirical Study of Industrial Code

Marek Vokáč

Simula Research Laboratory, NO-1325 Lysaker, Norway

Abstract

Software "Design Patterns" seek to package proven solutions to design problems in a form that makes it possible to find, adapt and reuse them. A common claim is that a design based on properly applied patterns will have fewer defects than more ad hoc solutions.

This case study analyzes the weekly evolution and maintenance of a large commercial product (C++, 500 000 LOC) over three years, comparing defect rates for classes that participated in selected Design Patterns to the code at large. We found that there are significant differences in defect rates among the Patterns, ranging from 63 % to 154 % of the average rate. We developed a new set of tools able to extract design pattern information at a rate of 3×10^6 lines of code per hour, with relatively high precision.

Based on a qualitative analysis of the code and the nature of the Patterns, we conclude that the Observer and Singleton patterns are correlated with larger code structures, and so can serve as indicators of code that requires special attention. Conversely, code designed with the Factory pattern is more compact and possibly less closely coupled, and consequently has lower defect numbers. The Template Method pattern was used in both simple and complex situations, leading to no clear tendency.

Key words: design patterns, defects, defect frequency, industrial code, case study, maintenance

3.1 Introduction

Software Design Patterns, as first formalized by Gamma *et al.* (1995), have become popular in the object-oriented software community. Some of the patterns have been incorporated into widely used architectures and frameworks. Examples of this are the Observer pattern in event-based user interfaces, and the Factory pattern in Microsoft COM, MFC and J2EE.

Common arguments for the use of Design Patterns often relate to defects; by designing an application with proper use of Design Patterns, we should reduce the number of defects (see for instance Guéhéneuc and Albin-Amiot (2001)). Others have claimed that positive consequences of using Design Patterns are additional flexibility and easier understanding of the design (Rising (1998); Buschmann *et al.* (1996); Larman (2001)), or expected reduced proneness to change (Bieman *et al.* (2003), though this study found the opposite, contrary to its expectations).

In this research, we wished to investigate the claims regarding reduced defects by determining the defect rates and Design Pattern usage of a large industrial software product. We performed a case study on a Customer Relationship Management (CRM) product. During the study we developed a tool for fast analysis and extraction of Patterns from C++ code. Having access to the complete history of the product for three years made it possible to analyze the defect rates and correlate them with the usage of Design Patterns. The defects were identified as a result of pre-release testing and post-release reporting by users.

The rest of the paper is organized as follows: Section 3.2 lists related work. Section 3.3 gives the goals, the choice of the studied software product and the pattern extraction tool used for the automated analysis. The statistical model and quantitative results are presented in Section 3.4. Threats to validity are addressed in Section 3.5. Finally, Section 3.6 summarizes the results, concludes and outlines future work.

3.2 Related work

Prechelt and Unger (1999); Prechelt *et al.* (2001) reported on two experiments on Design Patterns. The first experiment sought to test whether the presence of Design Patterns in program documentation had an effect on maintenance. The results were positive in the sense that maintenance was either faster, or introduced fewer errors, when the software was documented in terms of Design Patterns. Observer, Composite and Visitor were the tested patterns.

In the second experiment maintenance tasks were performed on "equivalent" versions of several programs, with one version containing Design Patterns and the other having a simpler, more straightforward structure. The results varied with the pattern used. Visitor was inconclusive, while Observer and Decorator resulted in less time being used for the maintenance tasks. Use of the Abstract Factory pattern had no significant effect.

This experiment was replicated by Vokáč *et al.* (2004), with the difference that a real programming environment was used instead of pen and paper as in the original experiment. Paid industry professionals were used as subjects. The results were similar for the Observer, Decorator and Abstract Factory patterns, while the Visitor and, to some extent, Composite patterns had strongly negative results, in that code derived from these patterns proved significantly harder to maintain, both in terms of time used and the number of errors.

A case study performed by Bieman *et al.* (2001) analyzed 39 versions of an evolving object-oriented software system. They found a strong relationship between class size and change frequency. Having taken this into account, they also found that classes that participate in Design Patterns are just as prone to change as other classes, and that proneness to change is positively correlated with reuse: classes that were the most reused were also the most change-prone. A later, more extensive study (Bieman *et al.*, 2003) mostly confirmed these findings, though with one significant contradiction.

A threat to the validity of the 2001 study (Bieman *et al.*, 2001) was the size and nature of the data set. The Singleton pattern was identified in 10 instances, Iterator in four and Factory Method and Proxy in only one instance each. The system under study evolved from 24 000 LOC and 199 classes to 32 000 LOC and 227 classes over the study span. Intermediate versions were not evaluated, but the total number of changes per class was counted. All changes were included, regardless of whether they were preventive, adaptive, perfective or corrective.

In the 2003 study (Bieman *et al.*, 2003), much larger software was studied, including NetBeans at 750 000 LOC, improving the external valid-

ity of the study. However, the identification of patterns was done manually, based on the documentation—on the explicit assumption that intentional patterns are the most interesting and that they will be documented by the developers. Patterns that are simply used by developers without being documented as such, will be missed by this approach.

Further case studies have been conducted by several groups (Schmidt and Stephenson, 1995; Neumann and Zdun, 2002; Chu *et al.*, 2000). The subjects of the studies are industrial systems of up to 30 000 LOC, but they address re-engineering or construction concerns, not maintenance over extended periods.

3.3 Case study goals, subject and conduct

The goal of this case study was to investigate the possible connection between the use of certain Design Patterns, and the error rates in the code. To achieve this, it was necessary to 1) choose the patterns to be studied; 2) obtain a suitable study subject, and 3) find or create a tool that can recover Design Patterns from code with sufficient speed and precision.

3.3.1 Design patterns

In this section, we describe the five patterns chosen for investigation, and the method by which they were chosen. For each pattern, we list the features that led us to expect different

The selection of Design Patterns was based on several criteria: the structure of the patterns, to investigate the effects of fundamentally different patterns; the occurrence of pattern names in academic literature; the occurrence of pattern names in relevant Web forums; and the occurrence of pattern names on the Web in general. The first criterion is considered the most important, while the remaining criteria we included to ensure that the chosen patterns had some practical relevance.

Academic literature was searched via the ISI Web of Knowledge, IEEE and ACM databases, and the results are shown in Table 3.1. Simply

	ISI		IEEE/ACM	
Pattern	Raw	Content	Raw	Content
Composite	4	4	42	15
Observer	8	6	9	8
Factory	5	4	16	9
Decorator	3	4	4	4
Adapter	1	3	5	4
Bridge	3	0	15	5
Singleton	2	2	4	3
Visitor	1	1	4	4
Proxy	1	0	5	4
Façade	3	1	2	2
Template Method	1	1	1	1
Sum	32	26	107	59

Table 3.1: Design Pattern ranks in academic literature. The "Raw" column lists the number of hits for the query, while the "Content" column lists the articles that were relevant.

counting the hits was not enough—the results of such searches can be quite imprecise. For instance, the word "'Composite" also matches "compositing" in the IEEE search engine, which leads to many false hits in the area of CASE tools and architecture papers. For the Web in general, a series of searches was done using Fast, AltaVista, Google, Yahoo, MSN and Lycos.[1] The search criteria were `"design patterns" software <pattern name>` for the candidate patterns listed below. The search target was the web in general—or more precisely, the subset of the Web included by each search engine.

We chose the following five patterns for our study: Singleton, because it may be tempting to use it simply as a replacement for global variables; Template Method, because it was known to be used extensively in the code we later wished to analyze; Decorator, because it can be a deceptively simple pattern that can hide quite complex behaviour; Observer, since it is central to the event-based architectures of several pro-

1. Excite, WebCrawler and MetaCrawler were excluded because they limit the number of hits to less than about 100, so they cannot be used for this kind of search.

gramming platforms (Microsoft Windows, .NET, and Java); and Factory, which has also become part of mainstream architectures. All of these patterns appeared to be of interest according to our surveys.

A Design Pattern, as described by Gamma *et al.* (1995), is a description, using prose and semi-formal diagrams, of a way to structure classes in a program. However, the ultimate expression of Design Patterns is in executable code, whether generated from a model or manually. It follows that the observed effects, in terms of defect rates or maintainability, are related to the patterns via the code structure. Our analysis of the expected effects is therefore founded on the kind of structure that the selected patterns prescribe.

The effects of some patterns on object-oriented software metrics have been described by Huston (2001). Patterns can influence several kinds of metrics: coupling measures, inheritance depth, and method counts. Huston concludes that the introduction of Design Patterns in general does not lead to code that exhibits metric values associated with high defect rates. This can at least be seen as not contradicting the claim that Design Patterns should lead to lower defect rates.

However, different patterns do have different effects and applicability. In the following paragraphs we briefly describe the structures and expectation connected to the five patterns chosen for the case study.

Factory Method—a simple structure

The Factory Method pattern is used in places where one of several possible subclasses (products) should be created, and the instantiating class (the client) cannot anticipate which subclass it should be. It may be that the knowledge does not belong in the client, or even that it is genuinely inaccessible to the client. Instead, it is located in a Factory Method.

We generally expect this pattern will lead to simple structures and relatively small amounts of code, without fundamentally influencing the architecture of the application.

Observer—a pattern for central structures

Observer, often implemented as Publish-Subscribe, is a pattern that specifies how a number of objects—observers—may be informed of changes to one object, the subject. Usually, the subject will be some kind of data-carrying object, and the observers will be views or processes that present or react to changes in the data. The classic Model-View-Controller pattern or one of its variants is often implemented using an Observer.

Both Observer and its related patterns (MVC, Publish-Subscribe) fundamentally influence the design of an application. If the subject is the applications' data model, and the observers are user interfaces, we expect not only that the implementation of the pattern to include a large amount of code, but also that that code will be fairly complex. The number of subjects is generally greater than one and is determined at runtime, making it much more difficult to arrive at a deterministic model of how the system behaves.

Singleton—global access to a single instance

In many ways, Singleton is similar to a global variable. Its scope may be limited by name spaces or similar constructions, but within the scope it is accessible to all. The main responsibility of the Singleton pattern is to ensure the existence of exactly one instance of the class in question.

Note that a Singleton carries state—otherwise it would not need an instance, and precise control over the number of instances. This leads us to expect that it will often be used for objects that should be accessible to a large number of dependent objects, if not always globally. A change to a Singleton will therefore tend to have a large impact. On the other hand, the presence of accessible singletons will often remove the need to pass parameters down long call structures, thereby avoiding needless dependencies.

Decorator—adding functionality in layers

This pattern is used to extend the functionality of a base object. Inheritance is appropriate when extensions are made in a hierarchical manner; Decorator is used where more than one extension can be active at any given time, and if the extension is to be made dynamically at run time.

Decorators can significantly reduce coupling in a system, since they are not visible from the outside of the pattern; clients simply call the method without ever being aware of a possibly large number of decorator objects inside. However, this can also make the system more difficult to analyze, since the actual call graph (including order of calls) cannot be determined by static analysis.

Template Method—replacing building blocks

The Template Method is suited to cases in which a high-level algorithm remains constant, but the underlying building blocks are subject to change.

The pattern is applicable to both large and small problems. The high-level task may be to fetch and process data from a database, and a building block might be the syntax for one specific DBMS. It could also be a sophisticated routing algorithm in which the building blocks are themselves algorithms composed of smaller, replaceable blocks, forming a whole tree of code.

3.3.2 The SuperOffice CRM5 product

The SuperOffice CRM5 product, made by SuperOffice ASA in Norway, was chosen as the study subject. It is a fairly large, mature commercial product. The company provided full access to the source code and its history, as well as the bug tracking database. The author was a senior member of the development team that created the studied version, and thus had good knowledge of the code.

The product is a Customer Relationship Management system, which

is used by companies to keep track of their customers, sales force diaries, activities and sales. It runs on Microsoft Windows and is a classic client/server application. The company has a heavy emphasis on user friendliness and consequently there is a lot of GUI code. The application is written entirely in C++, and was almost totally rewritten in the year 2000.

The application is sold in one standard version in 11 languages, and is installed at 11 000 customer sites. New versions were released approximately twice a year during the study period (2001-2003).

Metric	Value
Total lines	1 114 092
Lines of code (LOC)	505 367
Number of classes	2 047
Number of files	2 809
Number of methods	30 823
Declarative statements	150 685
Executable statements	194 625

Table 3.2: Descriptive metrics for SuperOffice CRM5

The code has been maintained and extended for the past three years by a stable team of developers, about half of whom were involved in writing the original code. A bug tracking system (TechExcel, 2004) was used to track reported defects in the code, whether found internally during formal pre-release testing, or at customer sites. The bug tracking system was partially integrated with the Control Version System (CVS (Perforce, Inc, 2004)). Changes to the code were checked into the version control system in transactions that generally corresponded to one functional change, whether corrective or otherwise.

Table 3.2 gives basic size metrics for the product, while Table 3.3 describes the evolution of the system during the 153-week period covered by the study.

Change type	No.	%	Lines added	changed	deleted
Corrective	1 619	31%	15 598	10 503	6 962
Other	3 562	69%	33 369	21 813	20 202
Total	5 181		48 967	32 316	26 164

Table 3.3: SuperOffice CRM5 evolution

3.3.3 Identifying design patterns in C++ code

Early in the study, we formulated the following goals for a pattern recovery tool:

- It must be possible to describe a structural signature and to extract from a C++ code base the set of classes that correspond to this signature. Given that some patterns result in complex structures, some programming effort is to be expected.
- The method must scale extremely well, and be able to handle amounts of code in the 10^8 LOC range with running times on the order of hours, or at least within a weekend. Preferably, the addition of new patterns should not force a re-run of the whole process.
- The input data should be in the form of "untreated" code files, i.e., whatever structure the source project is already in. Output should be in a form that is easily transferable to statistical packages for further analysis.

Existing tools found in the literature (Kramer and Prechelt, 1996; Florijn *et al.*, 1997; Bansiya, 1998; Antoniol *et al.*, 1998, 2001; Keller *et al.*, 1999; Schauer and Keller, 1998; Albin-Amiot *et al.*, 2001; Guéhéneuc and Albin-Amiot, 2001) have not been documented to possess the combination of speed, recovery and precision needed for our case study, with the possible exception of the work of Balanyi and Ferenc (2003)—which only appeared after our study was in progress.

Due to the lack of a satisfactory, off-the-shelf tool, we decided to construct our own. The tool was built using several sub-components to handle different stages of the process. During our case study, it performed satisfactorily. More details on the tool can be found in (Vokáč,

2005).

C++ code is parsed using a commercial parsing tool (Scientific Toolworks Inc., 2003), to create a database of metadata (lists of types, names, variables, methods, etc.,) and the relations between them, such as "method X calls method Y". The metadata is then transferred to an SQL Server database, indexed, and patterns are recovered through queries that correspond to the structural signature derived for each pattern. With simple index tuning, excellent performance was realized, while the use of SQL made it easy to debug the recovery process while working with the full data set.

In addition, if we wish to examine the evolution of the system over time, data extracted from CVS can be added to the metadata. By combining CVS data with the program metadata, the evolution of patterns over time can be traced, or pattern presence can be correlated with editing activities such as corrective maintenance.

Errors in the recovery of patterns from code would constitute a significant threat to the validity of this study. The subject is therefore treated in some detail in this section. There are two types of error that should be evaluated: false positive and false negative. A false positive occurs when a pattern is recognized, even if the classes concerned do not really conform to the pattern structure. A false negative is when classes do conform to the pattern, but are not recognized.

False positives are relatively easy to check, by inspecting the code that has been identified by the tool to determine whether it really does conform to a pattern. False negatives are, however, much more difficult to identify, since we would, in principle, have to identify all instances of all patterns in the code and then compare these to the output of the tool. For software of any significant size ($>$10 KLOC) this is not a realistic task. As a substitute for a total analysis, we can examine a random sample of the classes and determine the patterns present. This will give us an indication of the number of false negatives.

Generally, a precise and detailed specification of the structural signature of a pattern will reduce the number of false positives, but increase the risk of false negatives. The exact structural signature is tied to

language-specific features, and is also dependent on how stringently we interpret the structure of the pattern in question, viz. the discussions on inheritance depth and other parameters in papers by Florijn *et al.* (1997); Antoniol *et al.* (2001); Balanyi and Ferenc (2003).

One challenge is caused by the presence of preprocessor macros in the C and C++ languages. It is possible to code complicated declarations and structures in macros, which are then difficult to parse and analyze. While the problem is not completely intractable, as shown by Badros and Notkin (2000), the parsing tool we use has only limited preprocessor parsing support. This aspect must be taken into account when validating the use of our tool on any given set of software.

Since a Design Pattern is an informal specification of a recommended structure, it will be translated into program code differently in different projects. Any discussion of error rates must, therefore, be seen in relation to application of the method to a particular set of software items.

It is also necessary to define exactly what we mean by an instance of a certain Design Pattern. To take a simple example, consider Factory: one Factory class may have methods to create one or more Product classes. Should we count every combination of Factory and Product as an instance, or should one Factory class count as a single instance, regardless of the number of products? Similar situations occur in most patterns, since they specify relationships between multiple classes.

In our evaluations, we have adopted the simple definition, according to which one Factory class counts as a single instance. Similarly, we count one instance of the Observer pattern for every Subject; one Template Method for each template method; one Decorator for each Decorator class; and one Singleton for each Singleton class.

It was necessary to adjust the pattern-recovery tool for two particular patterns: Observer and Decorator. The Observer pattern can be implemented in two fundamentally different ways. The "classic" structure specifies that each Subject should keep track of its Observers directly, using a collection of object references. This introduces a fairly strong two-way coupling. An alternative approach is to use some kind of message broker to handle the relations between subjects and observers,

breaking the direct two-way coupling between a Subject and its Observers. This is the approach adopted globally in the CRM5 code, and the tool was adapted to this Subject/Broker/Observer structure.

For Decorator, a related problem exists, that of aggregation. The tool was adapted to the kind of aggregation generally used in CRM5 code.

False positives

The rate of false positives was determined by manually examining all detected instances of all patterns. The results for all patterns are given in Table 3.4.

Pattern	Instances	N_{false}	Error rate
Factory	53	8	15.1 %
Singleton	45	0	0.0 %
Observer	20	3	15.0 %
Template Method	163	2	1.2 %
Decorator	9	0	0.0 %

Table 3.4: False positives

Most of the false positives identified for the Factory pattern were classes that used inner (nested) classes. From the outside this looks like a Factory instance, since the inner class is created by the outer class and by nothing else. However, this usage does not correspond to the intent of Factory, so it was classed as a false positive. It is quite feasible to add the condition "product class must not be nested within factory class" to the structural signature for Factory in a refined version of the rules.

In the Observer cases, we are dealing with a "loosely coupled" version of Observer, in which all notifications are handled by a centralized message broker class. This differs somewhat from the classical, simple Observer pattern, in which every Subject keeps track of its Observers separately. There are other forms of interaction through the Message Broker than just those that accord with the Observer pattern, and the three false positives are such cases. Since the structure of the pattern is

already quite complex, further refinement is difficult without risking a greater number of false negatives.

The structure for the Template Method allows for multiple levels of inheritance, and does not require more than one call to an underlying, virtual primitive method to consider the caller a parent method. It is a matter of taste whether one would want to tighten the definition, i.e., to demand that there is more than one implementation of the primitive method, or more than one primitive method for each template method. The number of false positives would most probably decline, but the number of false negatives might increase.

Decorator has a somewhat problematic structure, in that it contains an aggregation (the set of Decorators for one decorated class) that can be implemented in many ways. The signature was adapted to the known kind of aggregation implementation in this code, and so we should not take the zero error rate as being guaranteed in a different setting. Also, false negatives are more probable for this pattern.

Singleton is a pattern that is fairly easy to recognize, and so the low false positive rate is as expected.

False negatives

To determine the actual rate of false negatives, we would need to evaluate all classes and find all cases in which a class participates in a pattern but has not been detected by the tool. With more than 2 000 classes this is not realistically possible.

Instead, we chose to evaluate a random sample of classes, to get an estimate of the false negative rate. From prior knowledge of the code (the author previously served as a developer and architect for the CRM5 product), a low rate of false negatives was expected. To calculate the necessary sample size, the following criteria were set:

Required power: 90%. Hypothesized proportion (false negative rate): 20%. Alternative proportion (rate to be tested for): 10%.

This yielded a required sample size of 109.[2] To guard against randomly choosing classes that are trivially small or otherwise nonrepresentative, the actual sample size was increased to 125.

Classes were chosen using a uniformly distributed random number generator. The chosen classes covered all major modules of the program. Figure 3.1 shows the distributions of class sizes, for the full system and the sample.

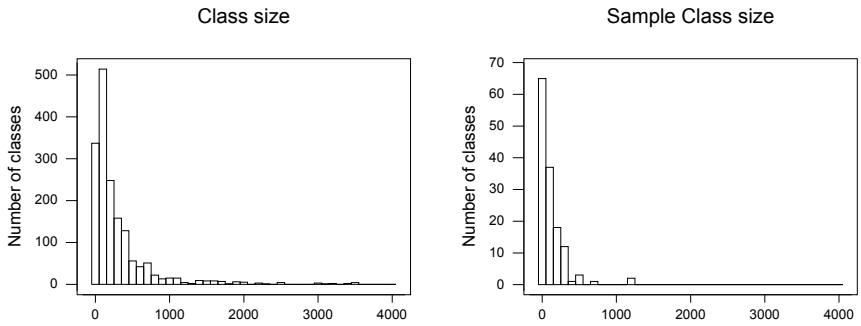

Figure 3.1: Histogram of class sizes, of the full system (left pane) and 125-class sample (right pane)

The 125 classes were inspected by the author together with a senior developer from the company. There were nine false negatives. Of these, one was part of a Decorator, one an Observer and the rest were Template Methods (six of the seven were actually part of the same instance of Template Method, missed due to macros in the code that hid the virtual method declarations).

The bounds for the false negative rate, as derived from these observations, are given in Table 3.5.

When interpreting these results, we must keep in mind that they apply to the SuperOffice CRM5 code only. Given the number of possible ways to implement the structure described by a pattern, the validity of the tool must be tested for each new coding style.

2. MiniTAB (MiniTAB, Inc, 2003) version 13.32 was used for all statistical calculations.

Pattern	N_{false}	95%	P
Factory	0	2.3 %	0.000
Singleton	0	2.3 %	0.000
Observer	1	3.7 %	0.000
Template Method	7	10.3 %	0.000
Decorator	1	3.7 %	0.000

Table 3.5: False negatives, from code sample

3.3.4 Extracting defects and design patterns from the CRM5 code

The presence of defects was determined by analysis of the CVS, where all the source code resides. This system was partially integrated with the Defect Tracking System used during the study period. A further textual analysis was made by the author of the comments in the CVS, to recover defects that did not have such direct links. The approach was validated by evaluating a sample drawn at random from the full set of code changes, to check that there were no wrong classifications.

The next step in the study was to obtain weekly snapshots of the complete source code, for the whole study period. Each snapshot reflects the state of the system at midnight between Saturday and Sunday, starting on February 4^{th}, 2001, when the system consisted of approximately 3 700 files totalling 90 MB of text. The snapshots grew slightly over time as code was added to the system; in sum, they consist of about 500 000 files and 14 GB of text.

The pattern extraction tool was applied to each snapshot, to parse all the code, load the metadata into the SQL Server, index it and extract Design Patterns. The whole process took slightly less than 26 hours on standard PC hardware.

3.4 Statistical model and quantitative results

This section presents the evolution and rationale behind our statistical model, and the purely quantitative results obtained from it.

The goal of our case study was to determine whether the presence of certain design patterns is correlated with the defect frequency of the code. Our raw data consist of the C++ code (here divided into classes), size metrics for each class, and indicators for whether or not the class participates in a pattern. The amount of code that is not a member of a class (global functions) is so small as to be negligible ($< 0.1\,\%$).

3.4.1 A simple model

To analyze the data, a binary logistic regression model was chosen (Kleinbaum, 1994). In this model, there are two possible outcomes of an observation, one of them termed an "event" or "success". The logistic equation is

$$G(\text{event}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \cdots \beta_n X_n \tag{3.1}$$

where G is the link function, which maps the interval $(0, 1)$ onto the real numbers. We use the standard logit function, $G(z) = \frac{1}{1+e^{-z}}$. The X's represent the effects we wish to measure, plus known or suspected confounding factors. An *event* $(z = 1)$ is defined as the occurrence of at least one corrective change to one class within one snapshot (week).

We define one indicator variable X_{Pattern} for each design pattern we wish to test for: X_{Obs}, X_{Sing}, X_{Decor}, X_{TM} and X_{Fact}. From both previous work (Bieman *et al.*, 2001) and intuition we know that code size is a possible confounding variable; it is reasonable to assume that there are more defects in a large class than in a small one. We also wish to test for any systematic trends in time, e.g., whether the defect rate varies in a linear fashion over time. We thus add the factors X_{Size} and X_{Week} to the model.

The estimates $\hat{\beta}_i$ are generally interpreted in terms of odds ratios, giving the change in odds for an event when the corresponding X_i changes by one unit. We therefore express code size in KLOC; adding one line to a class should not affect the number of defects much, but adding a thousand lines certainly would. The time variable is expressed as a week number, since we have one snapshot of the code per week; the

range is $(1 \ldots 153)$. The X_{Pattern} are indicator variables, with 0 denoting the absence, and 1 the presence, of a pattern for a given class.

Our data set consists of all the classes in the system, over the entire study period. This constitutes a total of 236 876 observations. While these are actually 153 repeated observations of 1550 classes[3], we can still consider them "independent" in the sense that we are considering the defect frequency relative to design patterns, size and time. Given the size of the system, the possible presence of a few classes with abnormally high defect rates should not have an undue influence.

An alternative method of analysis might have been to perform an Analysis of Variance, on a data set that consolidates all 153 snapshots. However, ANOVA is not tailored for a binary response variable, and the data material does not contain all combinations of all factors, which results in the model containing empty cells. Logistic regression is a natural choice for this kind of data.

This gives rise to the following model:

$$\begin{aligned} \frac{1}{1+e^{-z}} &= \beta_0 + \beta_O X_{\text{Obs}} + \beta_S X_{\text{Sing}} + \beta_D X_{\text{Decor}} \\ &\quad + \beta_T X_{\text{TM}} + \beta_F X_{\text{Fact}} \\ &\quad + \beta_K X_{\text{Size}} + \beta_W X_{\text{Week}} \end{aligned} \tag{3.2}$$

where

$$z = \begin{cases} 1 & \text{if a corrective change occurred} \\ 0 & \text{if no corrective change occurred} \end{cases}$$

Non-corrective changes are not counted in this model. There were 1 619 events out of a total of 236 876 observations. The β_i were estimated using MiniTAB Binary Logistic Regression, with the following results:

The primary results from a logistic regression are the odds ratios, whose interpretation is as follows: given a change of one unit in the

3. The discrepancy between the number of classes cited here and in Table 3.2 is caused by the elimination of undefined classes, templates, classes from standard system libraries and other library code that has not been maintained.

Coefficient		P	Odds Ratio	95% CI Lower	95% CI Upper
Constant	(β_0)	0.000			
Factory	(β_F)	0.000	**0.66**	0.54	0.81
Singleton	(β_S)	0.000	**2.69**	2.24	3.24
Observer	(β_O)	0.000	**1.53**	1.33	1.75
Template Method	(β_T)	0.002	**0.61**	0.44	0.83
Decorator	(β_D)	0.159	0.49	0.18	1.32
Size KLOC	(β_K)	0.000	**1.98**	1.85	2.11
Week	(β_W)	0.000	**0.99**	0.99	0.99

Log-Likelihood $= -9290.559$ Test that all slopes are zero: G $= 789.579$; DF $= 7$; P-Value $= 0.000$

Goodness-of-Fit Tests

Method	χ^2	DF	P
Pearson	97398	10^5	1.000
Deviance	13031	10^5	1.000

Table 3.6: Quantitative results from the fitting of the regression model in equation 3.2 to the observed data

underlying factor, and keeping all other factors unchanged, the odds ratio gives the change in probability for the occurrence of an event; in our case, the correction of a defect. As an example, an odds ratio of 0.5 for Factory would mean that code participating in the Factory pattern has one half the defect probability of all other code that has the same size and participation in other patterns.

This model yields several interesting results. First, four patterns have strongly significant odds ratios, but not in the same directions. The Factory and Template Methods are correlated with a lower defect frequency, while Observer and especially Singleton are correlated with a higher defect frequency.

Second, the Size effect is highly significant, as expected. Adding 1 000 LOC to a class roughly doubles the probability of a defect, all other circumstances being constant. Finally, there is a very slight downward

trend (odds ratio 1:0.99) in the number of defects over time. However, this trend is not considered large enough to invalidate the other results.

3.4.2 A full model including interactions

The model in the previous section only takes into account the main effects (the five patterns under study) and two possible confounders (size and time). However, this is too simple as there are more effects to consider.

First, the participation of classes in patterns is not a simple 1:1 or 1:0 relation. It is possible for a class to participate in more than one pattern; indeed, the occurrences of patterns and pattern combinations in the data material, as shown in Table 3.7, indicate that combinations must be taken into account.

Patterns	Occurrences	%
No Pattern	183 634	77.5 %
Factory	20 237	8.5 %
Singleton	3 331	1.4 %
Observer	16 061	6.8 %
Template Method	5 381	2.3 %
Decorator	1 513	0.6 %
Factory + Observer	612	0.3 %
Factory + Singleton	2 279	1.0 %
Observer + Singleton	2 390	1.0 %
Observer + Template	953	0.4 %
Factory + Observer + Singleton	485	0.2 %

Table 3.7: Frequencies of pattern occurrences, and percentage of code covered by the patterns

To take this into account, the model is recoded. Instead of using an individual indicator variable for each pattern, the pattern participation of each class is expressed as a combined PATTERN variable. The variable contains an 'F' if the class participates in Factory, an 'S' if it participates in Singleton, etc. A class that participates in both Factory and Singleton would have 'FS' as its PATTERN.

The regression is then re-run with Pattern as a factor, i.e., each distinct value is considered a separate coefficient. As before, the baseline is formed by those classes that do not participate in any pattern. We then obtain the results in Table 3.8.

		Odds	95% CI	
Coefficient	P	Ratio	Lower	Upper
Constant (No pattern)	0.000			
Week	0.000	**0.99**	0.99	0.99
Size (KLOC)	0.000	**1.90**	1.77	2.03
Factory	0.001	**0.68**	0.53	0.86
Singleton	0.000	**4.30**	3.43	5.40
Observer	0.000	**1.83**	1.57	2.13
Template Method	0.011	**0.63**	0.44	0.90
Decorator	0.170	0.50	0.19	1.34
Factory + Singleton	0.001	**2.00**	1.35	2.96
Factory + Observer	0.500	1.32	0.59	2.96
Observer + Singleton	0.000	**2.08**	1.43	3.04
Observer + Template Method	0.637	1.20	0.57	2.52
Factory + Observer + Singleton	0.145	1.82	0.81	4.09

Table 3.8: Quantitative results from the fitting of the recoded model with pattern combinations to the observed data.

We observe that two combinations yield significant odds ratios: Factory + Singleton and Observer + Singleton. This means that classes that participate simultaneously in both Factory and Singleton are twice as error-prone as classes that do not participate in any pattern (odds ratio 2.00); a similar result (odds ratio 2.08) is seen for the combination of Observer + Singleton.

However, there is another possible confounding factor to be taken into account: the possible interaction between size and pattern. This would be the case if certain patterns are correlated with a higher or lower size than other patterns or the classes in general; such a correlation would imply a collinearity between the pattern and the size coefficients.

We already expected that some patterns (Observer) would lead to

larger classes than others (Factory), viz. the discussion in Sections 3.3.1 and 3.3.1. The possibility of an interaction between pattern participation and code size must be assumed, and so we add interaction terms $X_{\text{Pattern}} X_{\text{Size}}$ to the model for each pattern.

3.4.3 Final model

The full set of Pattern $\times$ Size interactions yields a number of non-significant coefficients, and their presence disrupts the rest of the model. We therefore eliminate those interaction terms that are not significant; similarly, we eliminate the Pattern $\times$ Pattern terms from Table 3.8 that were not significant.

We thus obtain the following final model and results:

$$\begin{aligned}\frac{1}{1+e^{-z}} &= \beta_0 + \beta_O X_{\text{Obs}} + \beta_S X_{\text{Sing}} + \beta_D X_{\text{Decor}} \\ &\quad + \beta_T X_{\text{TM}} + \beta_F X_{\text{Fact}} \\ &\quad + \beta_K X_{\text{Size}} + \beta_W X_{\text{Week}} \\ &\quad + \beta_{SO} X_{\text{Sing}} X_{\text{Obs}} + \beta_{SK} X_{\text{Sing}} X_{\text{Size}} \\ &\quad + \beta_{OK} X_{\text{Obs}} X_{\text{Size}} \end{aligned} \tag{3.3}$$

where

$$z = \begin{cases} 1 & \text{if a corrective change occurred} \\ 0 & \text{if no corrective change occurred} \end{cases}$$

3.4.4 Interpretation of results

From the quantitative results in Table 3.9 we see that no significant correlation is detected for the Decorator and Singleton patterns. The remaining coefficients are significant, and the goodness-of-fit tests indicate that the model fits the data well.

The odds ratios must be interpreted with some care. The presence of a correlation does not by itself guarantee *causality*; for that we must go back to the code and perform a more qualitative analysis. In addition,

Coefficient		P	Odds Ratio	95% CI Lower	Upper
Constant	β_0	0.000			
Week	β_W	0.000	**0.99**	0.99	0.99
Size (KLOC)	β_K	0.000	**1.69**	1.53	1.87
Factory	β_F	0.000	**0.63**	0.51	0.77
Singleton	β_S	0.141	1.35	0.91	2.02
Observer	β_O	0.000	**1.55**	1.26	1.91
Template Method	β_T	0.048	**0.72**	0.52	1.00
Decorator	β_D	0.154	0.49	0.18	1.31
Singleton + Observer	β_{SO}	0.000	**0.32**	0.21	0.48
Singleton × Size	β_{SK}	0.000	**13.18**	6.29	27.61
Observer × Size	β_{OK}	0.009	**1.21**	1.05	1.40

Log-Likelihood = -9245.342 Test that all slopes are zero: G = 880.012; DF = 10; P-Value = 0.000

Goodness-of-Fit Tests

Method	χ^2	DF	P
Pearson	98299	10^5	1.000
Deviance	12941	10^5	1.000

Table 3.9: Quantitative results from the fitting of the final regression model in equation 3.3 to the observed data

interaction effects modify the interpretation of of the results. However, even at a quantitative level the results are interesting.

We can see that the systematic evolution in the number of defects with time was very slight, as the Week odds ratio is very close to 1. The estimated value of 0.99 indicates a small decrease in the defect ratio over time. Thus, while the defect rate falls significantly (P=0.000), the size of the effect is negligible. By contrast, adding 1 000 lines of code to a class increases the probability of defects by two thirds (ratio 1.69), except for the Observer and Singleton patterns, where there is an interaction effect to be taken into account.

For the Factory pattern, the expectation from Section 3.3.1 is supported.

The odds of a defect in a class involved in a Factory pattern (both the Factory class itself and its products) are less than two thirds (0.63) of the "background" defect rate. The insignificance of the Factory $\times$ Size and Factory $\times$ Singleton interactions (both dropped from the final model for lack of significance) increases our confidence that we are indeed looking at a true effect associated with the pattern—or rather, an *indication of the complexity of the situations in which this pattern was used*.

For Decorator, the pattern is not used much (one Window class has eight different, simple Decorators attached) and therefore the data are too sparse.

The Template Method, on the other hand, is used in many different contexts in the code, ranging from small and simple, to relatively deep and complex (3-4 levels inheritance and nested template functions). The wide spread is the cause of the rather weak results seen here, P=0.048. However, in the context of complexity and pattern use, the observation that there is only a weak relation is itself interesting. Template Method is a pattern that can hide a complex system, or be used in simple circumstances. Balaniy and Ferenc remarked that Template Method is a trivial pattern (Balanyi and Ferenc, 2003), but we see here that its use may vary from trivial to quite complex. The tendency observed here is that it does tend to lower the defect rate somewhat (an odds ratio of 0.72), so it is used more for the simpler, rather than complex, functions.

In the CRM5 system, the Singleton pattern is used for objects that are more or less global in scope, in the form of caches, utilities and repositories of application state. An example is a cache of user preferences; another is a set of interconnected objects that keep track of the GUI state (select panels, current data set, etc.). In the regression model we do not have a significant result (odds ratio 1.35, but P=0.141) for Singleton in isolation.

However, Singleton is a pattern that has two significant results in combination with other factors: Size, and the Observer pattern. Significant interaction factors mean that the effect is a function of the variable it interacts with, and the main coefficient β_S only yields the effect when the interacting variable is 0, i.e., Singletons that are *not* simul-

taneously Observers. We follow the interpretation given by DeMaris (1991), Hosmer and Lemeshow (2000) and Jaccard (2001), and calculate the odds ratio for Singleton *given simultaneous participation* in Observer as $\widehat{\text{OR}}_{SO} = \exp[\beta_S + \beta_{SO}]$ =0.43.

The explanation for the interaction is found by a qualitative analysis of the classes in question in the code. A set of seven small classes form a state machine that controls the global GUI state of the whole application. These classes are Singletons and simultaneously Observers of each other. They were carefully designed early on in the project, and since the underlying requirements have not changed, they have remained very stable. Thus, their defect rate is also small. This explains why Singleton and Observer tend towards higher defect rates when used on their own (odds ratios of 1.35 and 1.55 respectively), but have a significantly lower defect rate when used together. We would not expect this interaction effect to be generally valid for other programs.

Since size in KLOC is a continuous variable, the interpretation of the interaction between a pattern and size yields a set of odds ratios, one for each chosen value of the size. To determine proper size values and avoid unjustified extrapolation, we must look at the size distributions of the classes that participate in the Singleton and Observer patterns; the histograms are shown in Figure 3.2.

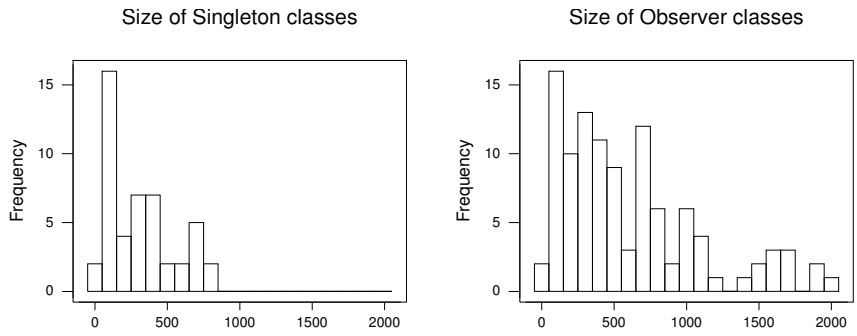

Figure 3.2: Histogram of class sizes, of the Singleton classes (left pane) and Observer classes (right pane)

The odds ratio for the expected defect rate, for a combination of a given pattern and a given code size is calculated as $\widehat{\text{OR}} = \exp[\beta_{\text{pattern}} +$

$\beta_{\text{interaction}} \times \text{KLOC}$]. Table 3.10 shows these odds ratios for Singleton and Observer, for code sizes that correspond to the actual classes in the CRM5 system.

KLOC	Singleton OR	Observer OR
0.1	1.75	1.58
0.25	2.58	1.63
0.5	4.91	1.71
1.0	17.82	1.88
1.5	—	2.07
2.0	—	2.28

Table 3.10: Odds ratios for Singleton and Observer, for representative class sizes

We can observe that classes that participate in the Singleton pattern are very sensitive to size. This is partly explained by presence of the small, stable classes mentioned above. The same effect applies to the Observer $\times$ Size interaction, with the difference that there are more instances of the Observer pattern than the Singleton pattern (6.7 %, vs. 1.8 % of the total code, from Table 3.7). This is why the interaction is significant, yet does not invalidate the main effect related to Observer.

The odds ratio for Observer alone is 1.55 (this corresponds to setting the size value to 0), which supports our expectation from Section 3.3.1: Observer is a relatively complicated pattern that is used in situations with coupling between multiple, nontrivial classes. The higher than average defect frequency is as expected.

3.5 Threats to validity

In a study such as the present one, there are multiple threats to validity, both internal and external. Internal validity is concerned with the consistency of the measurements, appropriate use of tools and methods. External validity concerns the degree to which the data and results are transferable outside the particular context of the study.

One threat arises from the use of an automated tool to recover design patterns from code, an inherently imprecise and difficult process. The tool was, therefore, tested and validated, and the results from Section 3.3.3 indicate that, in the current study, the tool is sufficiently reliable to be used. This means that it has acceptably low false positive and false negative rates, when applied to the code in this case study. The classification of false positive and false negatives was performed by the author together with a senior developer from the company, to reduce the risk of incorrect classifications.

The choice of the patterns to be analyzed was based partly on their known structure and expected effects, and partly on their popularity as determined from surveys of academic literature, discussion groups and the Web in general.

A second threat is related to the software chosen for analysis, in that it should be of sufficient size and complexity for patterns and defects to occur often enough to give statistically valid results. As discussed in Section 3.4, the pattern Decorator did not fulfill this criterion. No strictly quantitative conclusions can therefore be drawn regarding Decorator from the current material. The spread of values for Template Method is an interesting and valid result, given the high number of occurrences, despite its lack of simple quantitative significance.

The choice of statistical model and its content must also be considered. Given the size of the data set, a detailed analysis and classification of each defect is beyond the scope of the study. With a binary outcome (defect/no defect) and indicator variables for the presence or absence of patterns, logistic regression is a natural model.

In addition to main effects and two confounders (size and time), all pattern $\times$ pattern interaction terms that were actually present in the data were evaluated. Further interaction terms for patterns versus size were included, based on intuition and the experience of other workers in the field, especially Bieman *et al.* (2001; 2003), who found a strong size effect. In our case, a size effect was found for two patterns (Observer and Singleton), linked to a special set of small, stable classes in the code.

The external validity of the study is mainly limited by two factors: the

applicability of the tool to other coding styles in other software, and the usage of design patterns elsewhere. These threats reflect some of the inherent problems of case studies, where numerous cultural and technical factors are impossible to characterize completely. The partly inconsistent results reported by Bieman *et al.* (2003) are an example. To some extent these threats can only be addressed by extending the knowledge base with studies that span multiple systems, cultures and styles.

The software itself is a long-lived industrial product of considerable size, thus small size or unrealistically simple design should not be a threat. It was designed by a team that was aware of Design Patterns in general, yet had not studied them in great detail. It is to be hoped that this is representative of commercial software designed with patterns.

The studied defects were identified both as a result of internal pre-release testing, and from reports from actual users. The study span of three years ensured that there was, in fact, time to incorporate user feedback, causing measurable changes to appear in the code.

As argued in Sections 3.3.1 and 3.3.1, we believe it is inherent in the nature of the Factory and Observer patterns that the former should be used in simpler, less tightly connected code than the latter. The observed results support this observation. However, only further observations of other systems will support generalization outside the studied domain.

The applicability of the pattern extraction tool is fairly easy to check for any other software system, by performing a validation similar to the one done in Sections 3.3.3 and 3.3.3. If no major adjustments need to be made to the recognition algorithms, we may conclude that the tool is more generally applicable. However, the general applicability of the tool, or lack of it, is not a significant factor in the applicability of the results of the present case study, for which the tool was extensively validated.

3.6 Summary, conclusions and future work

3.6.1 Summary of results

We wished to determine whether there is any systematic correlation between the occurrence of certain Design Patterns and defect frequency in an industrial product. To analyze the source code, we designed and implemented a tool capable of extracting information about the presence of selected design patterns from C++ code. The tool analyzed 76×10^6 LOC in less than 26 hours. The patterns were selected based on surveys of the Web and academic literature, as well as the suitability for evaluating the tool.

False positive errors (identification of a pattern where none exists) were determined by inspecting each pattern occurrence identified by the tool, and varied from 0 % to 15 %. False negative errors (missing a pattern) were estimated statistically from a random sample of classes, and varied from 2 % to 10 %. These rates compare favourably with other work in the field. The combination of speed and precision allows the analysis of large software products over time.

The tool was then applied to a Customer Relationship Management product, consisting of more than 500 KLOC. Snapshots were obtained for each week in a three-year period, and analyzed using a logistic regression model. The response variable was the presence of a defect in the code, and the model terms were the presence of the five patterns, size and week number, and interaction terms.

Significant correlations were found for the Factory, Observer and Template Method patterns. Code related to Factory had a lower defect rate (63 %) than the code in general, while Observer was correlated with higher defect rates (155 %). In the case of Singleton and Observer there were also significant interactions with size, which supports the hypothesis that these patterns tend to be used in complex areas, with more code and higher defect frequencies. Defect frequency increased with size for these patterns, especially for Singleton.

Template Method occurred many times in many different contexts, so the spread of defect frequencies was large and no strong, single conclu-

sion can be drawn. However, this illustrates the many uses to which this pattern can be put, and the different effects it may have. The Decorator pattern did not occur often enough to yield statistically significant results.

3.6.2 Conclusions

The usage of Design Patterns has been cited as a way to make good designs easier to develop, even for less experienced developers. As a corollary, since good design is presumed to lead to lower defect rates and other benefits, the use of patterns is also expected to lead to lower defect rates.

However, we find that the reality, at least in the case of the software studied, is not quite that simple. Well-known Design Patterns have widely different sizes, complexities and applicability, so that the use of patterns by itself is no guarantee of few defects.

In any non-trivial software product, there will be areas that have an irreducible, significant complexity. Unless they are designed and maintained very carefully, such areas will have higher defect rates than the average in the product.

We believe that some patterns, notably Singleton and Observer in our study, tend to be associated with such complexity. Thus, even the "proper" use and implementation of these patterns may not be enough to reduce the defect rate to the general average. However, the applicability of these patterns can serve as a useful warning sign to the developers: if Observer is found to be the proper solution, then that area of the software is probably inherently complex and warrants above-average effort in its design and implementation. As good design resources are always at a premium, it is hoped that these conclusions may help developers to target their resources on the most beneficial areas.

3.6.3 Future work

Future work is envisioned as an iterative process between tool development and code analysis. In its current state our design pattern extrac-

tion tool is fast enough to be used on large projects, but it is necessary to validate it on different kinds of program, from different development groups. The Open Source community should be a good source of projects to analyze, and work is already underway to perform similar analyses on a large number of projects. This should result in an improved tool in terms of applicability to different coding styles. Extending the number of patterns analyzed is also relevant.

With an improved tool, an analysis similar to the present case study can be performed, to see whether our results have general applicability. The existing material can also be reanalyzed for any new patterns added to the tool.

Further study is possible by calculating relevant metrics for the code, to see how the presence of design patterns correlates with trends in the metrics and defect frequencies.

Acknowledgement

The author would like to extend sincere thanks to SuperOffice ASA and Director of Development Guttorm Nielsen, for granting unrestricted access to the CRM5 source code. Prof. Erik Arisholm made major contributions to the statistical modelling, and Prof. Dag Sjøberg provided valuable comments on the style and structure of the paper.

Bibliography for paper 3

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y. G., Jussien, N., 2001. **Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together**. In: **ASE 2001: 16th Annual International Conference on Automated Software Engineering**. IEEE CS Press, San Diego, CA, USA, pp. 26–29.

Antoniol, G., Casazza, G., Di Penta, M., Fiutem, R., 2001. **Object-Oriented Design Patterns Recovery**. Journal of Systems and Software 59 (2), 181–196.

Antoniol, G., Fiutem, R., Cristoforetti, L., 1998. **Using Metrics to Identify Design Patterns in Object-Oriented Software**. In: **Metrics 1998: Fifth International Software Metrics Symposium, 1998.** IEEE Computer Society, Bethesda, Maryland, USA, pp. 23–34.

Badros, G. J., Notkin, D., 2000. **A Framework for Preprocessor-Aware C Source Code Analyses**. Software-Practice & Experience 30 (8), 907–924.

Balanyi, Z., Ferenc, R., 2003. **Mining Design Patterns from C++ Source Code**. In: **ICSM'03: International Conference on Software Maintenance**. IEEE Computer Society, Amsterdam, The Netherlands, pp. 305–315.

Bansiya, J., June 1998 1998. **Automating Design-Pattern Identification**. Dr. Dobb's Journal 23 (6), 20–2, 24, 26, 28.

Bieman, J., Jain, D., Yang, H., 2001. **OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study**. In: **ICSM 2001: IEEE International Conference on Software Maintenance, 2001**. IEEE Computer Society, Firenze, Italy, pp. 580–589.

Bieman, J., Straw, G., Wang, H., Munger, P., Alexander, R., 2003. **Design Patterns and Change Proneness: An Examination of Five Evolving Systems**. In: **METRICS '03: Ninth International Software Metrics Symposium, 2003.** IEEE Computer Society, Sydney, Australia, pp. 40–49.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. **Pattern-Oriented Software Architecture**. Wiley, Chichester, ISBN: 0 471 95869 7.

Chu, W. C., Lu, C. W., Shiu, C. P., He, X. D., 2000. **Pattern-Based Software Reengineering: A Case Study**. Journal of Software Maintenance—Research and Practice 12 (2), 121–141.

DeMaris, A., 1991. **A Framework for the Interpretation of First-Order Interaction in Logit Modeling**. Psychological Bulletin 110 (3), 557–570.

Florijn, G., Meijers, M., van Winsen, P., 1997. **Tool Support for Object-Oriented Patterns**. In: **ECOOP '97: European Conference on Object-**

Oriented Programming. Vol. 1241 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Heidelberg, pp. 472–495.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Boston, MA, USA, ISBN: 0201633612.

Guéhéneuc, Y.-G., Albin-Amiot, H., 2001. **Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects**. In: **TOOLS 39: 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001**. Santa Barbara, CA, USA, pp. 296–305.

Hosmer, D. W., Lemeshow, S., 2000. **Applied Logistic Regression**, 2nd Edition. John Wiley & Sons Inc., New York, USA, ISBN: 0471356328.

Huston, B., 2001. **The Effects of Design Pattern Application on Metric Scores**. Journal of Systems and Software 58 (3), 261–269.

Jaccard, J., 2001. **Interaction Effects in Logistic Regression**. Quantitative applications in the social sciences. Sage Publications, Thousand Oaks, CA, USA, ISBN: 0761922075.

Keller, R., Schauer, R., Robitaille, S., Pagé, P., 1999. **Pattern-Based Reverse-Engineering of Design Components**. In: **ICSE '99: 1999 International Conference on Software Engineering**. ACM Press, Los Angeles, CA, USA, pp. 226–235.

Kleinbaum, D. G., 1994. **Logistic Regression : A Self-Learning Text**. Statistics in the Health Sciences. Springer-Verlag Heidelberg, New York, ISBN: 0-387-94142-8.

Kramer, C., Prechelt, L., 1996. **Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software**. In: **Third Working Conference on Reverse Engineering, 1996**. IEEE Computer Society, Monterey, CA, USA, pp. 208–215.

Larman, C., 2001. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process**, 2nd Edition. Prentice Hall, Upper Saddle river, NJ, USA, ISBN: 0130925691.

MiniTAB, Inc, 2003. **MiniTab 13.32**. URL `http://www.minitab.com`

Neumann, G., Zdun, U., 2002. **Pattern-Based Design and Implementation of An XML and RDF Parser and Interpreter: A Case Study**. In: **ECOOP '02: 16th European Conference on Object-Oriented Programming**. Vol. 2374 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, University of Mlaga, Spain, pp. 392–414.

Perforce, Inc, 2004. **Perforce Software Configuration Management System**. URL `http://www.perforce.com/`

Prechelt, L., Unger, B., 1999. **Methodik und Ergebnisse einer Experimentreihe über Entwurfsmuster**. Informatik - Forschung und Entwicklung 14 (2), 74–82.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., Votta., L. G., 2001. **A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions.** IEEE Transactions on Software Engineering 27 (12), 1134–1144.

Rising, L., 1998. **The Patterns Handbook**. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0521648181.

Schauer, R., Keller, R., 1998. **Pattern Visualization for Software Comprehension**. In: **IWPC '98: 6th International Workshop on Program Comprehension, 1998**. pp. 4–12.

Schmidt, D., Stephenson, P., 1995. **Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms**. In: **ECOOP '95: European Conference on Object-Oriented Programming**. Vol. 952 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Århus, Denmark, pp. 399–423.

Scientific Toolworks Inc., 2003. **Understand for C++**. URL `http://www.scitools.com/`

TechExcel, 2004. **DevTrack Defect Tracking Tool**. URL `http://www.techexcel.com/products/devtrack/dtoverview.html`

Vokáč, M., 2005. **A Tool for Recovering Design Patterns from C++ Code, and its Application in a Case Study**. Journal of Object Technology, To appear July/August 2005.

Vokáč, M., Tichy, W., Sjøberg, D. I. K., Arisholm, E., Aldrin, M., 2004. **A Controlled Experiment Comparing the Maintainability of Programs Designed with and Without Design Patterns: A Replication in a Real Programming Environment**. Empirical Software Engineering 9 (3), 149–195.

Paper 4

An efficient tool for recovering design patterns from C++ code

This paper has been accepted for publication in the Journal of Object Technology, and is tentatively scheduled to appear in July 2005. The text included here is the finally accepted version.

An efficient tool for recovering Design Patterns from C++ Code

Marek Vokáč

Software Engineering Department, Simula Research Laboratory, Oslo, Norway

Abstract

Design Patterns are informal descriptions of tested solutions to recurring problems. Most design tools have little or no support for documenting the presence and usage of patterns in code. Reverse engineering is therefore often required to recover Design Patterns from code in existing projects. Knowledge of what Design Patterns have been used can aid in code comprehension, as well as support research.

Since pattern descriptions are abstract and informal, they offer no algorithmic translation into concrete code. Some patterns prescribe class structures that are easy to recognize, while others lead to structures that are difficult or impossible to recognize.

This work presents a tool that can recover five different design patterns from C++ code with high precision and at a speed of 3×10^6LOC/hr. This makes it suitable for analysis of large (multi-millon LOC) systems.

4.1 Introduction

Software Design Patterns, as first formalized by Gamma *et al.* (1995), have become popular in the object-oriented software community. Some of the patterns have been incorporated into widely used architectures and frameworks. Examples of this are the Iterator pattern in the C# language (coupled to the `foreach` keyword), the Observer pattern in event-based user interfaces, and the Factory pattern in Microsoft COM, MFC and J2EE.

Recovery of Design Patterns from existing code is important in several situations. Code maintenance should be made easier if any patterns used during the design can be recovered. Also, empirical research on the effects of using Design Patterns is severely limited if it cannot make use of existing code bases.

Hopefully, the emergence of integrated development environments that fully support UML modelling and pattern application will reduce the need for reverse-engineering tools; however, maintenance of code designed without such support will remain significant for many years.

To support our ongoing research into correlations between the use of Design Patterns and defect frequency, we needed a tool to extract patterns from a large amount of C++ code. A survey of academic literature uncovered a number of recovery tools, but none of them were documented to be able to handle the patterns and code sizes we needed. We therefore created and validated our own tool for this purpose.

The tool looks for structural signatures, i.e., class and method structures that result from the implementation of certain design patterns. We developed a semi-formal, graphic notation to describe Design Pattern structures in greater detail than the original diagrams given by Gamma *et al.*, but not so rigid as to overspecify and thereby miss recovery of implementations that are not identical to the "ideal" structure.

This paper is organized as follows: Section 4.2 discusses existing tools, as documented in the academic literature. Pattern structures and our diagrammatic notation for expressing them are described in Section 4.3. The goals and construction of our tool are in Section 4.4, and its performance, recovery and precision on a 500 000 LOC system are in Section 4.5. Section 4.6 concludes.

4.2 Existing tools and related work

In this section we summarize related pattern-extraction tools. Our starting point is executable code, not UML designs or other specifications.

The justification for this starting point is as follows. A Design Pattern is a description, using prose and semi-formal diagrams, of a way to structure classes in a program. However, the ultimate expression of Design Patterns is in executable code, whether generated from a model or by hand, and the code is the ultimate reference (as opposed to UML

models or other documents, which tend to become out of date if not used with good tool support, or rigorously maintained). Our survey of related work is thus restricted to tools that use code as their input.

It is in the very nature of Design Patterns that they are abstract, general prescriptions of solutions. Their translation into actual code necessarily involves judgement, and is not a task that can be performed mechanically without regard to context. This is especially true when analyzing existing code, the design of which was inspired by patterns yet did not have "compliance" with pattern specifications as its goal. This means that we should not expect any tool to recover all patterns with 100% precision.

An early work was by Kramer and Prechelt (1996). Patterns were drawn in an OMT design tool and translated into Prolog rules; source code was parsed using the Paradigm Plus tool and converted into Prolog facts. Then, queries were run to determine what facts matched the rules, ie., what patterns were present in the code.

The parsing tool had significant limitations. It did not extract information that would have been useful, such as whether a method is a constructor, or whether a class is abstract or concrete. Further, the tool looked only at header files, and thus had no information on the function call hierarchy. This made recovery of patterns more difficult, since essential parts of the signature of a pattern that depended on these concepts could not be expressed. Nevertheless, the tool achieved reasonable recall and precision rates on source code of moderate size (150–350 classes).

Florijn *et al.* (1997) constructed a tool that was integrated in a Smalltalk environment, which supported development at several abstraction levels, including that of Design Pattern. With this tool, it is possible to create new classes as instances of patterns, connect existing classes with patterns and roles, and check whether pattern invariants are being upheld by classes in the code. The real-life test example cited involves about 150 classes.

Bansiya (1998) presented a tool in 1998 that used the Microsoft MSVC compiler to parse the code, and relied on the "browse information"

database generated. The tool seems to have been based on a structural rule-based matching method, but there is little information on its precision, or its ability to handle large systems.

Antoniol *et al.* used a different approach (Antoniol *et al.*, 1998). The code was analysed in terms of tuples of classes and their relations, and metrics were used to reduce the number of candidate classes and avoid the combinatorial explosion.

The metrics calculated were the number of attributes and operations, further divided into public, private and protected; the number of association, aggregation and inheritance associations for each class; and the total numbers of attributes, methods and relations.

Using the metrics, classes were eliminated that did not show the "right" signature for the pattern in question, such as inheritance or associations that are part of the given patterns' structure. In the final stage, the exact pattern signature was sought among the classes that survived the metrics selection process.

Their method was tested on public-domain and industrial software in the 5 000–50 000 LOC range. It performed well in terms of speed (minutes), but did not achieve high precision. In the industrial software analyzed, there were so few pattern instances that it was difficult to judge the precision of the process.

The method has been further developed and was last presented in 2001 (Antoniol *et al.*, 2001); however, the precision of the system is still quite low (3%–50%).

A potential weakness is the fact that some metrics, such as inheritance, are not reliable indicators of a pattern structure. For instance, the Template Method pattern specifies a base class with a template method, and concrete subclasses that override and implement the primitive operations. However, this whole hierarchy might well be embedded in a larger context, so that the abstract base class is itself a subclass of one or more classes. Thus, it would be incorrect to conclude that a class has to be at the top of the hierarchy to be at the root of the Template Method pattern. However, it is correct to require that the subclass in the Tem-

plate Method really be a subclass; though it may be deeper than a direct subclass of the base class.

Keller *et al.* (1999) designed a pattern extraction mechanism for use within a larger reverse-engineering system. Based on the structure of the pattern, they constructed an appropriate query that searched their metadata repository for corresponding occurrences. Performance figures are not given, but one of the systems tested consisted of more than 470 000 lines of code. This work is a continuation of earlier work by Schauer and Keller (1998).

An interesting point is that there are some patterns that are difficult or impossible to recover, because their structural signature is weak or variable. The Bridge pattern is cited as an example; while the original definition of Bridge specifies a certain combination of inheritance and aggregation associations, in practice these are not always followed. Relaxing the criteria to accommodate this causes large numbers of false positives, while keeping them strict, means that design constructs intended to work like Bridge are missed. We believe this is an inherent property of the pattern, rather than of any particular approach to recovery.

Albin-Amiot *et al.* (2001) have proposed and implemented in prototype a system that searches for, and recognizes, pattern signatures in Java code. Their system relies on a constraint solver, which attempts to solve the problem given by matching the actual code structure to the structure of the design pattern. Their system can recognize partial or distorted implementations and can even recommend possible refactorings.

They do not state running times or give examples of the application of their system to non-trivial systems. However, they state that they intend to test their system on the package JHOTDRAW; a companion paper (Guéhéneuc and Albin-Amiot, 2001) states that this package "... contains more than 125 classes and identifies several design patterns".

Perhaps the most promising method to date for design pattern recovery from large-scale projects is that of Balanyi and Ferenc (2003). They use a reverse engineering framework to convert C++ code into metadata

(termed Abstract Semantic Graph), and express patterns in an XML-based language. They then perform a multi-step algorithm to identify candidate class structures, match them to the pattern descriptions, and filter out mismatches. One of their major contributions is to look at information from function bodies, such as function calls and object creations, in addition to the more traditional static structure.

They seem to be the only group so far that has tested a method on million-line code collections, and they give running times for extraction of different patterns. In two large projects of 1 200 KLOC and 1 500 KLOC size, they found about 440 and 520 pattern instances in total, in five and nine hours' running time. However, the time spent on code parsing is not given. They cite fairly low rates of false positives (falsely indicating the presence of a pattern), but do not give any evaluation of false negatives (failing to identify a pattern that is actually present).

4.3 Pattern structures and descriptions

As a first level of abstraction from concrete code, we adopted an entity-reference model, similar to the class/relation tuples used by Antoniol *et al.* (1998). An *entity* is anything that is not a language keyword or operator, i.e., any named class, variable, method, macro or parameter. A *reference* links two entities, and is of a certain type, such as "Calls", "Is declared by" or "Overrides". Entity types also express attributes such as "virtual" or "public". During the parsing step, the code to be analyzed is reduced to a set of entities interconnected by multiple references.

Using these two concepts, we constructed a graphic notation to define Design Patterns at a sufficient level of detail for the analysis. In our view, formal, rigorous definitions as used by France *et al.* (2004), are not suitable in this context, since our aim is to be able to recover patterns that have been applied imprecisely. Further, the "rigorous" application of a pattern may simply not be the best design decision in every case, and we expect software designers to exercise judgement.

In its first version, our tool recovers the patterns Observer, Decorator, Factory, Singleton and Template Method. Our concepts for pattern structures, and the structural signature left imprinted in the final code, are discussed for the Template Method and Observer patterns. They are good illustrations of both simple and more complex problems. We should note that during most of its running time, our tool performs parsing and preparatory indexing that is independent of the actual patterns to be extracted. Adding new patterns is thus fairly easy, and does not require the whole parsing process to be re-run.

4.3.1 Template Method

This is a relatively simple Design Pattern. It is used in situations where the major flow of a process or algorithm is given and is reusable, but there may be differences in the detailed steps. In this case, the algorithm is implemented in a base class, and calls overridable methods for the detail steps. These methods may be abstract or have a default implementation in the base class. Derived classes provide their own implementations of these methods as required, thus customizing the algorithm to their particular needs.

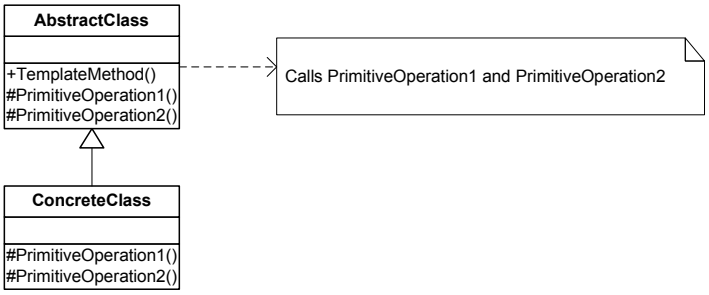

Figure 4.1: Structure of Template Method, from Gamma *et al.* (1995).

Gamma *et al.* use an informal, UML-like notation with explanatory prose to describe the structure of this design pattern, as shown in Figure 4.1. We see that it involves an abstract class and a concrete class, and one or more "primitive" operations that are called by the template method in the base class; these methods are overridden in the concrete class.

Some elements of this structure diagram should not be taken too literally. For instance, the inheritance may span multiple levels; there may be more than one template method in the abstract class, and concrete classes will not always override all of the primitive operations.

Our translation into the entity/reference model modifies and formalizes the structure to take into account these properties.

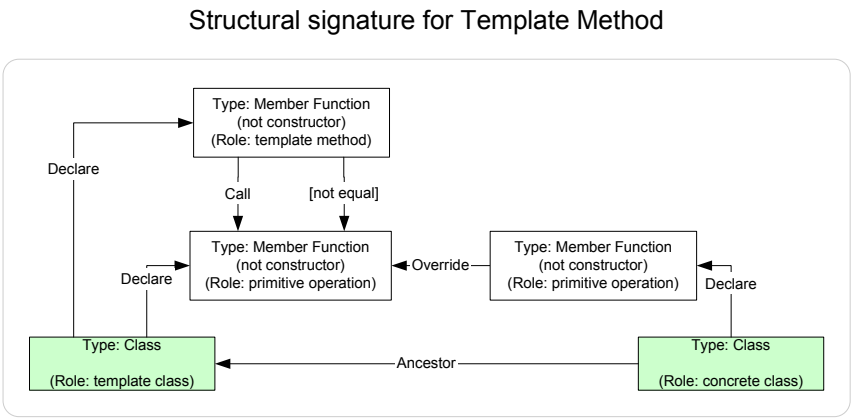

Figure 4.2: Structural signature of Template Method

In our notation, boxes represent entities such as classes or methods, and arrows between the boxes represent references. The top line of text in a box denotes the entity type, and the bottom line denotes the role that the entity plays in the pattern. Restrictions are placed in the middle of the box. Similarly, the text on a reference defines the reference kind.

The signature diagram for the Template Method pattern is shown in Figure 4.2, and expresses the expected structural signature in a more complete way than the original notation. Our diagrams can be translated by hand or semi-mechanically into executable queries against an entity-reference database derived from actual code.

The two main entities in the signature diagram are the template class and the concrete class (shaded boxes). The template class is a (possibly indirect) ancestor of the concrete class, and may itself be a subclass of some other class (this is not specified and thereby not restricted). The template class declares at least two member functions, which have the

additional restrictions of not being the constructor. We require a "call" reference from the template member function to one or more primitive operation functions, and we require the functions to be distinct.

The concrete class must declare one or more member functions that are also not constructors, and which override the template class methods that act as primitive operators.

4.3.2 Observer

The Observer pattern illustrates how it is possible to implement a design pattern in at least two, radically different ways. The core concept is simple: a class (the observer) may observe changes to another class (the subject), and react to those changes in some way; there may be multiple types and instances of Observers for any Subject.

The differences in implementation are related to the way in which notifications from subjects to observers are implemented. In the classic, closely coupled model, described in the original pattern, each observer keeps track of its subjects and directly notifies them of changes. The structure described in Gamma *et al.* (1995) and shown in Figure 4.3 portrays the closely coupled model.

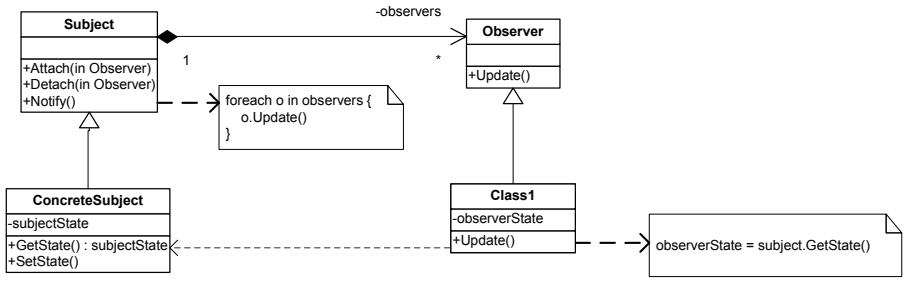

Figure 4.3: Structure of Observer, from Gamma *et al.* (1995).

In this model, there are base classes for the concepts of Subject and Observer, and concrete classes derived from them. The subject class maintains references to its current set of observers, and the observers call the subject directly to retrieve information. It is explicitly specified that the subject state is not part of the notification message, but must be

retrieved separately.

However, the Observer pattern is also known under the name "Publish-Subscribe", which points us to a different, loosely coupled model for its implementation. In this model there is a third party, a broker, that keeps track of subjects, observers and notifications. The concrete subject class is relieved of this task and does not have to be a subclass of a common "subject" base. The observer class will, in most instances, still have some relation to the subject (it is, after all, interested in what happens to the subject) but its registration interaction will be with the broker.

There are other ways of communicating as well. Buschmann *et al.* (1996) describes several possible schemes. The Reactor pattern (Schmidt, 1994) describes an inter-process variant with Singleton multiplexers and demultiplexers in the sending and receiving processes. A similar scheme, though without the multiplexing, appeared in the OMG Event Service Specification Object Management Group (1995). It involves a proxy publisher and a proxy subscriber to hide the process boundary, and an event channel to transfer the notifications. Varying degrees of buffering, asynchronicity and further decoupling are also possible.

The tightly coupled and loosely coupled observer implementations have quite different signatures, as seen in Figure 4.4. The loosely coupled model involves a total of nine entities and 12 references, while the tightly coupled model uses six entities and seven references. Most importantly, we will not recognize a tightly coupled observer while looking for a loosely coupled one, and vice versa.

This range of potential implementations is a fundamental and intended strength of the concept of Design Patterns; the designer is able to implement a pattern in the way that best fits the context and problem at hand. However, by the same token, that same range of potential implementations makes it much more difficult to reconstruct patterns from code, unless there are some "standard" implementations whose structural signatures will match most of the actual usage. As a result, it must be made clear regarding a particular reverse-engineering tool what variants of patterns it is designed to recover.

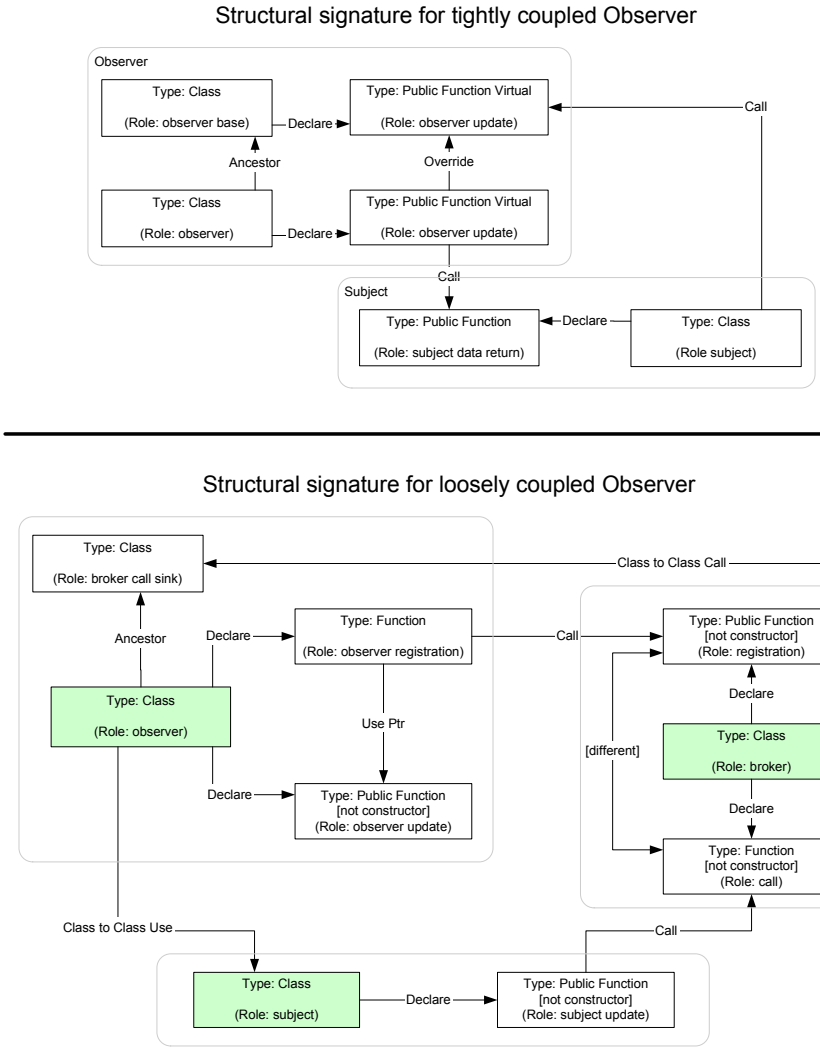

Figure 4.4: Structural signatureof Observer, tightly and loosely coupled.

4.3.3 Language-specific features

Different languages implement concepts in different ways. One concept that is central to many Design Patterns is that of aggregation, generally implemented as a collection; a set of objects or references to objects. The Composite pattern is a typical example, where each composite object may contain or reference any number of child objects. To recognize a Composite, we therefore need to start by recognizing a collection or aggregation relation.

In C++ in general, this is almost impossible. A C++ pointer can point to one object, or it can point to an array of objects; but there is no way to know which without analyzing the code in great detail.

Before the advent of the Template mechanism and the Standard Template Library SGI (2004), every developer or group had to implement its own data structures, so there was no generally accepted standard (unlike Java and its Collections hierarchy of objects). To make matters even worse, macros in C++ can be used to perform almost any textual substitution, making parsing almost impossible. An example of this is a case where a Container base class is inherited through a macro that defines a subclass; correctly parsing and recognizing this as a collection is beyond most tools (Badros and Notkin, 2000).

In other languages the situation is simpler. Java has a well-defined set of container classes Sun Microsystems, Inc (2004), and also does not have the fine division between embedded objects, references and pointers present in C++. Thus, it is much easier to detect collections with reasonable accuracy in Java.

In C# there is an `IEnumerator` interface in the standard library set, which encourages developers to implement their own iterator concepts wherever appropriate. It is coupled to the `foreach` keyword in the language Microsoft, Inc (2004), yielding an easily recognized, simple and elegant syntax for traversal of any array or collection.

Our tool and the exact structural signature for the chosen patterns were designed in the context of C++, and to some extent adjusted for the programming style used in the software analyzed in the case study.

However, the underlying concepts are transferable to other, similar languages.

4.4 Tool goals and design

We formulated the following goals for a pattern recovery tool:

- It must be possible to describe a structural signature of a pattern, and to extract from a C++ code base the set of classes that correspond to this signature.
- The tool must be flexible, because some design patterns have complicated signatures that are not easily expressible as simple rules. It must be reasonably easy to express a structure, so that the specification can be checked, revised and debugged.
- The tool must scale extremely well, and be able to handle amounts of code in the 10^8 LOC range with running times in the order of hours, or at least within a weekend. Preferably, much of the processing time should be spent on data preparation that is independent of what patterns are being sought, so that the addition of new patterns does not force a re-run of the whole process. If possible and when necessary, the method should lend itself to optimization using parallel hardware (storage, CPU) or clustering.
- The input data should be in the form of "untreated" code files, i.e., whatever structure the source project is already in. Output should be in a form that is easily transferable to statistical packages for further analysis.

4.4.1 Tool construction

Due to the lack of a satisfactory, off-the-shelf tool, we decided to construct our own. The tool was built using several sub-components to handle different stages of the process. During our research, it satisfied all the goals, with the exception of parallel processing, which was not pursued due to lack of suitable hardware.

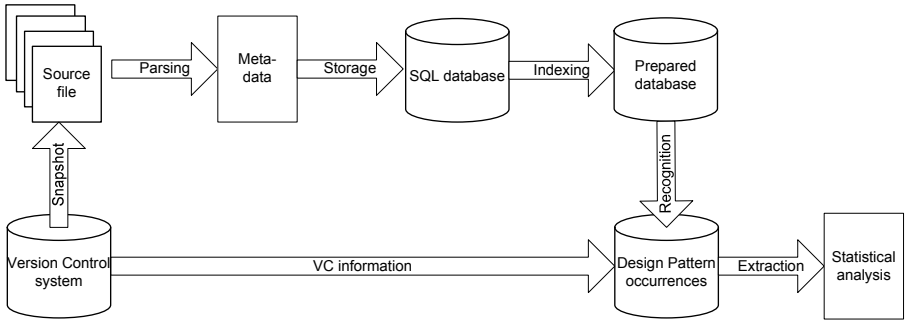

Figure 4.5: Outline of the structure extraction tool. The process starts at the lower left.

Figure 4.5 shows an outline of how our tool is constructed. The first stage consists of extraction of code snapshots from a Version Control System (VCS). If we only want to perform a single analysis of a system, this stage can be performed manually (and does not require the presence of a VCS at all). However, for analyses of trends over time we need to take multiple snapshots, one for each time point we wish to analyze.

The resulting snapshot is a collection of C++ source files, both header and class body. The method makes no assumptions as to the location of classes or correspondence between classes and files, but since the VCS generally works at the file level, analysis becomes simpler if the convention of "one class, one file pair" is followed.

The source files are parsed using a commercial tool called UNDERSTAND FOR C++ (Scientific Toolworks Inc., 2003). This tool is flexible and scalable, and parses C++ code into metadata. Its main shortcoming is that it does not currently handle templates (generics), and this causes some problems later on with design patterns that rely on collections.

The output from UNDERSTAND is a file, in proprietary format, that contains mainly two kinds of data: entities and references. An entity is any named concept that is not a keyword, for instance a class, variable, type or file. A reference is a link between two entities, such as "declares", "calls", or "dereferences". Both entities and references are classified

into predefined kinds.

The "storage" stage of the processing transfers the entity and reference data from the proprietary UNDERSTAND format into an SQL database, without materially changing the data. The database contains the following: (i) tables that correspond to the entity and reference concepts, and (ii) supporting tables for entity and reference kinds, and links to files and metrics.

The tables are then indexed. Insertion of large amounts of data is much faster if there are no indexes, so index generation is postponed until data loading is completed. During the indexing process, some extra reference kinds are computed in addition to those generated by Understand. To optimize performance, the database schema is slightly denormalized by converting some relations (such as whether an entity is a class member function) into attributes directly in the entity table.

At this stage, the database is ready to perform recognition of structural design patterns. The recognition is actually done by a series of SQL statements designed to look for the given structure; a structural signature translates quite readily into one or more `select` statements. Complicated or irregular structures may be recovered by chaining multiple SQL statements in a stored procedure. This provides more expressive power than single statements, and also provides an opportunity for optimizing performance where necessary. The results are stored in intermediate tables.

Metadata is also extracted from the VCS, in the form of information about submitted changes. This information is added to the database with the code metadata, which enables us to combine it with the class structure and see how it evolved.

Finally, the results are condensed and transferred to a statistical package for further analysis. An example of such analysis is to generate indicator variables for each pattern, and use logistic regression to look for correlations between changeability and pattern membership for classes.

4.5 Tool performance, recovery and precision

4.5.1 Performance

The tool was used to analyze a Customer Relationship Management system, written in C++. Table 4.1 gives some size metrics for the product.

Metric	Value
Total lines	1 114 092
Lines of code (LOC)	505 367
Number of classes	2 047
Number of code files	2 809
Number of methods	30 823
Declarative statements	150 685
Executable statements	194 625

Table 4.1: Descriptive metrics for SuperOffice CRM5

The system consisted of approximately 3 700 files (including scripts, resources, graphics, etc), totalling 90 MB. However, as our study intended to analyze the system's evolution over time, a total of 153 weekly snapshots were extracted from the VCS. The snapshots grew slightly over time as code was added to the system; in sum, they consisted of about 500 000 files totalling 14 GB.

The pattern extraction tool was applied to each snapshot, to parse all the code, load the metadata into the SQL Server, index it and extract Design Patterns. Typical running times are shown in Figure 4.6. Each snapshot had its own, preallocated empty database in the server. Analysis of all the snapshots, 76×10^6 LOC, took slightly less than 26 hours on a 2.8 GHz PC with 4 GB of RAM (max. 800 MB actually in use) and standard IDE disk drives.

Table 4.2 summarizes the frequency of occurrence of the patterns in the code. Participation of a class in a pattern is not a simple 1:0 or 1:1 relation, as it is quite possible for a class to participate in multiple patterns. Our tool considers each pattern separately, but since the entities that

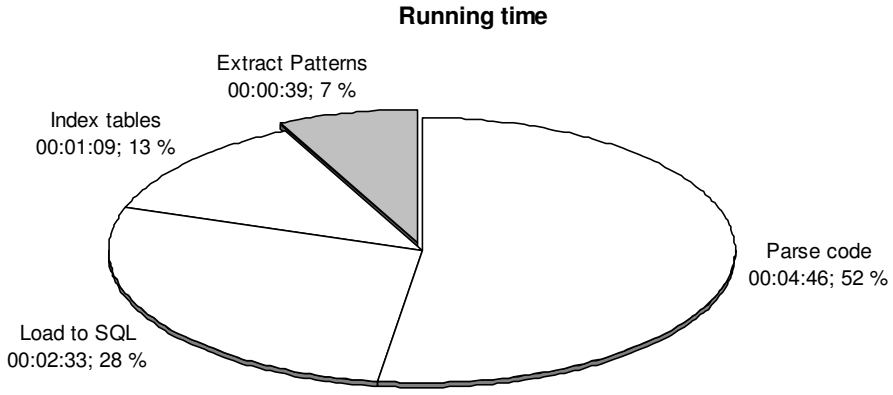

Figure 4.6: Length and phase distribution of running times for the Pattern recognition tool, on 500 000 LOC.

represent classes in the metadata have unique identifiers, it is easy to identify classes that participate in more than one pattern.

Patterns	Occurrences	%
No Pattern	183 634	77.5 %
Factory	20 237	8.5 %
Singleton	3 331	1.4 %
Observer	16 061	6.8 %
Template Method	5 381	2.3 %
Decorator	1 513	0.6 %
Factory + Observer	612	0.3 %
Factory + Singleton	2 279	1.0 %
Observer + Singleton	2 390	1.0 %
Observer + Template	953	0.4 %
Factory + Observer + Singleton	485	0.2 %

Table 4.2: Frequencies of pattern occurrences, and percentage of code covered by the patterns

4.5.2 Error rates

Since a Design Pattern is an informal specification of a recommended structure, it will be translated into program code differently in different projects. Any discussion of error rates must, therefore, be seen in relation to the application of the method to a particular set of software artifacts.

It is also necessary to define exactly what we mean by an instance of a certain Design Pattern. Consider Factory as an example: one Factory class may have methods to create one or more Product classes. Should we count every combination of Factory and Product as an instance, or should one Factory class count as a single instance, regardless of the number of products? Similar situations occur in most patterns, since they specify relationships between multiple classes.

In our evaluations, we have adopted a simple definition, according to which one Factory class counts as a single instance. Similarly, we count one instance of the Observer pattern for every Subject; one Template Method for each template method; one Decorator for each Decorator class; and one Singleton for each Singleton class.

It was necessary to adjust the pattern-recovery tool for two particular patterns: Observer and Decorator. The Observer pattern can be implemented in two different ways, as discussed in section 4.3. The "classic" structure specifies that each Subject should keep track of its Observers directly, using a collection of object references. An alternative approach is to use a generalized message broker to handle the relations between subjects and observers, and this is the approach adopted globally in the studied code. The tool was adapted to this Subject/Broker/Observer structure.

For Decorator, a related problem exists, that of aggregation (see section 4.3.3). The tool was adapted to the kind of aggregation generally used in studied code.

4.5.3 False positives

The rate of false positives was determined by manually examining all detected instances of all patterns. The results for all patterns are given in Table 4.3.

Pattern	Instances	False	Error rate
Factory	53	8	15.1 %
Singleton	45	0	0.0 %
Observer	20	3	15.0 %
Template Method	163	2	1.2 %
Decorator	9	0	0.0 %

Table 4.3: False positives

Most of the false positives identified for the Factory pattern were classes that used inner (nested) classes. From the outside this looks like a Factory instance, since the inner class is created by the outer class and by nothing else. However, this usage does not correspond to the intent of Factory, so it was classed as a false positive. It is feasible to add the condition "product class must not be nested within factory class" to the structural signature for Factory in a refined version of the rules.

In the Observer cases, we are dealing with a "loosely coupled" version of Observer, in which all notifications are handled by a centralized message broker class. This differs somewhat from the classical, simple Observer pattern, in which every Subject keeps track of its Observers separately. There are other forms of interaction through the Message Broker than just those that accord with the Observer pattern, and the three false positives are such cases. Since the structure of the pattern is already quite complex, further refinement is difficult without risking a greater number of false negatives.

The structure for the Template Method allows for multiple levels of inheritance, and does not require more than one call to an underlying, virtual primitive method to consider the caller a parent method. It is a matter of taste whether one would want to tighten the definition, i.e., to demand that there is more than one implementation of the primitive

method, or more than one primitive method for each template method. The number of false positives would most probably decline, but the number of false negatives might increase.

Decorator has a somewhat problematic structure, in that it contains an aggregation (the set of Decorators for one decorated class) that can be implemented in many ways. The signature was adapted to the known kind of aggregation implementation in this code, and so we should not take the zero error rate as being guaranteed in a different setting. Also, false negatives are more probable for this pattern.

Singleton is a pattern that is fairly easy to recognize, and so the low false positive rate is as expected.

4.5.4 False negatives

To determine the actual rate of false negatives, we would need to evaluate all classes and find all cases in which a class participates in a pattern but has not been detected by the tool. With more than 2 000 classes this is not realistically possible.

Instead, we chose to evaluate a random sample of classes, to get an estimate of the false negative rate. Judging from prior knowledge of the code (the author previously served as a developer and architect for the studied product), a low rate of false negatives was expected. To calculate the necessary sample size, the following criteria were set:

Required power: 90%. Hypothesized proportion (false negative rate): 20%. Alternative proportion (rate to be tested for): 10%.

This yielded a required sample size of 109.[1] To guard against randomly choosing classes that are trivially small or otherwise nonrepresentative, the actual sample size was increased to 125.

Classes were chosen using a uniformly distributed random number generator. The chosen classes covered all major modules of the program. Figure 4.7 shows the distributions of class sizes, for the full system and the sample.

1. MiniTAB (MiniTAB, Inc, 2003) version 13.32 was used for all statistical calculations.

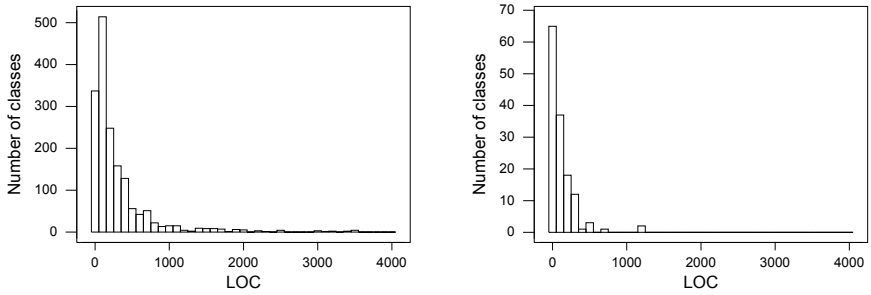

Figure 4.7: Histogram of class sizes, of the full system (left pane) and 125-class sample (right pane)

Out of the 125 classes inspected, a total of nine were false negatives. Of these, one was part of a Decorator, one an Observer and the rest were Template Methods (six of the seven were actually part of the same instance of Template Method, missed due to macros in the code that hid the virtual method declarations).

From these observations, we calculate the upper bound of a 95% confidence interval for the proportion of false negatives. The results are given in Table 4.4.

Pattern	N_{false}	95% CI	P
Factory	0	2.3 %	0.000
Singleton	0	2.3 %	0.000
Observer	1	3.7 %	0.000
Template Method	7	10.3 %	0.000
Decorator	1	3.7 %	0.000

Table 4.4: False negatives, from code sample

When interpreting these results, we must keep in mind that they apply to the studied code only. Given the number of possible ways to implement the structure described by a pattern, the validity of the tool must be tested for each new coding style.

4.6 Summary and Future work

We have successfully built and validated a tool that efficiently recovers selected Design Patterns from C++ code. The tool was used to analyze a 500 000 LOC commercial system. During processing of historical data from the VCS, the tool evaluated a total of 76×10^6 LOC.

The tool is based on descriptions of structural signatures associated with the chosen Design Patterns. The signatures are described using semi-formal diagrams, which can be translated into queries mechanically, or hand-coded in the case of complicated or irregular structures. The code to be analyzed is parsed into metadata using a commercially available tool, and this metadata is placed in a relational database where the queries are executed.

The tool has been empirically validated to have a high rate of recovery (few false negatives) and precision (few false positives). However, validation has been performed only on one major set of code. Since Design Patterns are usually not mechanically applied and translated into concrete code, it is necessary to revalidate the tool when applying it to new software.

Most of the tool's running time is spent on tasks that are independent of the specific patterns recovered. This, combined with its speed, means that adding further patterns is relatively straightforward, and can be done while using a full code base and not just trivial examples.

We plan to pursue the tool in four possible directions:

- Add more patterns—many of the patterns in Gamma *et al.* (1995) are candidates, though not patterns such as Bridge, which seem inherently to leave a very imprecise signature.
- Define a UML Profile for our pattern description notation, and use a code generator to transform pattern structure diagrams into SQL queries
- Validate the tool on more software—the Open Source community should be a good source of nontrivial C++ software.
- Extend the tool to other languages—since the first stage is to translate from C++ into an abstract entity-reference model, an

equivalent parser for other languages can be substituted. While it may be unrealistic to find another parser with exactly the same output, the entity-reference model is so general that it should be easy to transform other formats into it.

Bibliography for paper 4

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y. G., Jussien, N., 2001. **Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together**. In: **ASE 2001: 16th Annual International Conference on Automated Software Engineering**. IEEE CS Press, San Diego, CA, USA, pp. 26–29.

Antoniol, G., Casazza, G., Di Penta, M., Fiutem, R., 2001. **Object-Oriented Design Patterns Recovery**. Journal of Systems and Software 59 (2), 181–196.

Antoniol, G., Fiutem, R., Cristoforetti, L., 1998. **Using Metrics to Identify Design Patterns in Object-Oriented Software**. In: **Metrics 1998: Fifth International Software Metrics Symposium, 1998.** IEEE Computer Society, Bethesda, Maryland, USA, pp. 23–34.

Badros, G. J., Notkin, D., 2000. **A Framework for Preprocessor-Aware C Source Code Analyses**. Software-Practice & Experience 30 (8), 907–924.

Balanyi, Z., Ferenc, R., 2003. **Mining Design Patterns from C++ Source Code**. In: **ICSM'03: International Conference on Software Maintenance**. IEEE Computer Society, Amsterdam, The Netherlands, pp. 305–315.

Bansiya, J., June 1998 1998. **Automating Design-Pattern Identification**. Dr. Dobb's Journal 23 (6), 20–2, 24, 26, 28.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. **Pattern-Oriented Software Architecture**. Wiley, Chichester, ISBN: 0 471 95869 7.

Florijn, G., Meijers, M., van Winsen, P., 1997. **Tool Support for Object-Oriented Patterns**. In: **ECOOP '97: European Conference on Object-Oriented Programming**. Vol. 1241 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Heidelberg, pp. 472–495.

France, R., Kim, D.-K., Ghosh, S., Song, E., 2004. **A UML-Based Pattern Specification Technique**. IEEE Transactions on Software Engineering 30 (3), 193–206.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Boston, MA, USA, ISBN: 0201633612.

Guéhéneuc, Y.-G., Albin-Amiot, H., 2001. **Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects**. In: **TOOLS 39: 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001**. Santa Barbara, CA, USA, pp. 296–305.

Keller, R., Schauer, R., Robitaille, S., Pagé, P., 1999. **Pattern-Based Reverse-Engineering of Design Components**. In: **ICSE '99: 1999 International Conference on Software Engineering**. ACM Press, Los Angeles, CA, USA, pp. 226–235.

Kramer, C., Prechelt, L., 1996. **Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software**. In: **Third Working Conference on Reverse Engineering, 1996**. IEEE Computer Society, Monterey, CA, USA, pp. 208–215.

Microsoft, Inc, 2004. **C# Programmer's Reference: Foreach, in**. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrftheforeachstatement.asp`

MiniTAB, Inc, 2003. **MiniTab 13.32**. URL `http://www.minitab.com`

Object Management Group, 1995 1995. **CORBAServices: Common Object Services Specification**.

Schauer, R., Keller, R., 1998. **Pattern Visualization for Software Comprehension**. In: **IWPC '98: 6th International Workshop on Program Comprehension, 1998**. pp. 4–12.

Schmidt, D. C., 1994. **Reactor: An Object Behavioural Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching**. In: Coplien, J. O., Schmidt, D. C. (Eds.), **PLoP 94**. Addison-Wesley, pp. 529–545.

Scientific Toolworks Inc., 2003. **Understand for C++**. URL `http://www.scitools.com/`

SGI, 2004. **Standard Template Library Programmer's Guide**. URL `http://www.sgi.com/tech/stl/`

Sun Microsystems, Inc, 2004. **Java API Documentation: Interface Collection**. URL `http://java.sun.com/j2se/1../docs/api/java/util/Collection.html`

Complete bibliography

The following bibliography is the superset of all the section bibliographies, including the Introduction and the four peer-reviewed papers. References here are given in alphabetical order, regardless of where they appeared.

Adrion, W. R., 1992. **Research Methodology in Software Engineering**. In: Tichy, W. F., Habermann, N., Prechelt, L. (Eds.), **Dagstuhl Workshop on Future Directions in Software Engineering**. ACM SIGSOFT, Schloss Dagstuhl, pp. 36–37.

Aeinehchi, N., 2002. **Do NOT Use WebSphere Unless You are BLUE**. URL `http://www.theserverside.com/reviews/thread.jsp?thread_id=13639`

Agerbo, E., Cornils, A., 1998. **How to Preserve the Benefits of Design Patterns**. In: **OOPSLA '98: Conference on Object Oriented Programming Systems Languages and Applications**. Vol. 33 of SIGPLAN Notices. ACM Press, Vancouver, British Columbia, Canada, pp. 134–143.

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y. G., Jussien, N., 2001. **Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together**. In: **ASE 2001: 16th Annual International Conference on Automated Software Engineering**. IEEE CS Press, San Diego, CA, USA, pp. 26–29.

Alexander, C., 1977. **A Pattern Language: Towns, Buildings, Construction**. Center for Environmental Structure. Oxford University Press, New York, ISBN: 0195019199.

Alexander, C., 1979. **The Timeless Way of Building**. Center for Environmental Structure. Oxford University Press, New York, ISBN: 0195024028.

Alexander, C., 1985. **The Production of Houses**. Oxford University Press, New York, ISBN: 0195032233.

Alexander, C., Hirshen, S., Ishikawa, S., Coffin, C., Angel, S., 1969. **Houses Generated by Patterns**. Center for Environmental Studies, Berkely.

Almaer, D., 2002. **Making a Real World PetStore, TSS Newsletter #31**. URL `http://www.theserverside.com/resources/article.jsp?l=PetStore`

Alur, D., Crupi, J., Malks, D., 2001. **Core J2EE Patterns**. Prentice-Hall, Upper Saddle River, NJ, USA, ISBN: 0130648841.

Ambler, S. W., 1998. **Process Patterns**. The Press Syndicate of the University of Cambridge, Cambridge, United Kingdom, ISBN: 0521645689.

Anderson, B., 1992. **Towards An Architecture Handbook**. In: **OOPSLA '92: Conference on Object Oriented Programming Systems Languages and Applications**. Vol. 27 of SIGPLAN Notices, Issue 10. ACM Press, Vancouver, British Columbia, Canada, pp. 109–113.

Anonymous, 2002. **Pattern Forms**. URL `http://c2.com/cgi/wiki?PatternForms`

Antoniol, G., Casazza, G., Di Penta, M., Fiutem, R., 2001. **Object-Oriented Design Patterns Recovery**. Journal of Systems and Software 59 (2), 181–196.

Antoniol, G., Fiutem, R., Cristoforetti, L., 1998. **Using Metrics to Identify Design Patterns in Object-Oriented Software**. In: **Metrics 1998: Fifth International Software Metrics Symposium, 1998.** IEEE Computer Society, Bethesda, Maryland, USA, pp. 23–34.

Apache Jakarta Project, 2003. **STRUTS Home Page**. URL `http://jakarta.apache.org/struts/`

Arisholm, E., 2001. **Empirical Assessment of Changeability in Object-Oriented Software**. Phd thesis, University of Oslo, Norway.

Arisholm, E., Sjøberg, D., 2004. **Evaluating the Effect of a Delegated Versus Centralized Control Style on the Maintainability of Object-Oriented Software**. IEEE Transactions on Software Engineering 30 (8), 521–534.

Arisholm, E., Sjøberg, D., Carelius, G. J., Lindsjørn, Y., 2002a. **A Web-Based Support Environment for Software Engineering Experiments**. Nordic Journal of Computing 9 (4), 231–247.

Arisholm, E., Sjøberg, D., Carelius, G. J., Lindsjørn, Y., 2002b. **SESE: An Experiment Support Environment for Evaluating Software Engineering Technologies**. In: **NWPER 2002: Tenth Nordic Workshop on Programming and Software Development Tools and Techniques**. Copenhagen, Denmark, pp. 81–98.

Arisholm, E., Sjøberg, D. I. K., Jørgensen, M., 2001. **Assessing the Changeability of Two Object-Oriented Design Alternatives—A Controlled Experiment**. Empirical Software Engineering 6 (3), 231–277.

Badros, G. J., Notkin, D., 2000. **A Framework for Preprocessor-Aware C Source Code Analyses**. Software-Practice & Experience 30 (8), 907–924.

Baer, W. C., 2002. **The Institution of Residential Investment in Seventeenth-Century London**. Business History Review 76 (Autumn 2002), 515–552.

Balanyi, Z., Ferenc, R., 2003. **Mining Design Patterns from C++ Source Code**. In: **ICSM'03: International Conference on Software Maintenance**. IEEE Computer Society, Amsterdam, The Netherlands, pp. 305–315.

Bansiya, J., June 1998 1998. **Automating Design-Pattern Identification**. Dr. Dobb's Journal 23 (6), 20–2, 24, 26, 28.

Beck, K., 1987. **Using a Pattern Language for Programming**. In: Kerth, N. L., Hogg, J., Stein, L., Porter, H. H. (Eds.), **OOPSLA'87: Addendum to the Proceedings**. ACM Press, Orlando, Florida, USA, p. 16.

Beck, K., 1999. **Extreme Programming Explained: Embrace Change**. Addison Wesley, Boston, MA, USA, ISBN: 0201616416.

Beck, K., Beedle, M., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D., 2001. **Agile Manifesto**. URL `http://www.agilemanifesto.org/`

Bernus, P., Nemes, L., 1996. **A Framework to Define a Generic Enterprise Reference Architecture and Methodology**. Computer Integrated Manufacturing Systems 9 (3), 179–191.

Berry, C., Carnell, J., Juric, M., Kunnumpurath, M., Nashi, N., Romanosky, S., 2002. **J2EE Design Patterns Applied**. Wrox Press Ltd, Hoboken, NJ, USA, ISBN: 1861005288.

Bieman, J., Jain, D., Yang, H., 2001. **OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study**. In: **ICSM 2001: IEEE International Conference on Software Maintenance, 2001**. IEEE Computer Society, Firenze, Italy, pp. 580–589.

Bieman, J., Straw, G., Wang, H., Munger, P., Alexander, R., 2003. **Design Patterns and Change Proneness: An Examination of Five Evolving Systems**. In: **METRICS '03: Ninth International Software Metrics Symposium, 2003.** IEEE Computer Society, Sydney, Australia, pp. 40–49.

Black, E., 2002. **IBM and the Holocaust**. Time Warner Paperback, ISBN: 0751531995.

Boehm, B., 1986. **A Spiral Model of Software Development and Enhancement**. ACM SIGSOFT Software Engineering Notes 11 (4), 14–24.

Booch, G., 1993. **Object-Oriented Analysis and Design**, 2nd Edition. Pearson

Education, Upper Saddle River, NJ, USA, ISBN: 0805353402.

Borchers, J., 2001. **A Pattern Approach to Interaction Design**. John Wiley & Sons, Hoboken, NJ, USA, ISBN: 0471498289.

Brooks, F. P. J., 1987. **No Silver Bullet: Essence and Accidents of Software Engineering**. IEEE Computer 20 (4), 10–19.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. **Pattern-Oriented Software Architecture**. Wiley, Chichester, ISBN: 0 471 95869 7.

Christensen, L. B., 2001. **Experimental Methodology**, 8th Edition. Allyn & Bacon, Boston, MA, USA, ISBN: 0-205-30832-5.

Chu, W. C., Lu, C. W., Shiu, C. P., He, X. D., 2000. **Pattern-Based Software Reengineering: A Case Study**. Journal of Software Maintenance—Research and Practice 12 (2), 121–141.

Ciancarini, P., Tolksdorf, R., Vitali, F., Rossi, D., Knoche, A., 1998. **Coordinating Multiagent Applications on the WWW: A Reference Architecture**. IEEE Transactions on Software Engineering 24 (5), 362–375.

Coplien, J., Schmidt, D., 1995. **Pattern Languages of Program Design**. Addison Wesley, Boston, MA, USA, ISBN: 0201607344.

DeGrace, P., Stahl, L. H., 1991. **Wicked Problems, Righteous Solutions**. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, ISBN: 013590126X.

DeMaris, A., 1991. **A Framework for the Interpretation of First-Order Interaction in Logit Modeling**. Psychological Bulletin 110 (3), 557–570.

Diggle, P., Liang, K., Zeger, S., 1994. **The Analysis of Longitudinal Data**, 2nd Edition. Oxford University Press, Oxford, United Kingdom, ISBN: 0198524846.

Ditzel, C., 2003. **Charles's Corner: Java Technology Pointers**. URL `http://java.sun.com/jugs/pointers.html`

Douglass, B. P., 2002. **Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems**. Addison-Wesley, Boston, MA, USA, ISBN: 0201699567.

Efron, B., Tibshirani, R. J., 1993. **An Introduction to the Bootstrap**. Monographs on Statistics and Applied Probability. Chapman & Hall, London, United Kingdom, ISBN: 0-412-04231-2.

Ekström, U., 2000. **Design Patterns for Simulations in Erlang/OTP**. Master's thesis, Uppsala University, Sweden.

Feynman, R., 1997. **Surely You're Joking, Mr Fenyman**. W. W. Norton & Com-

pany, New York, USA, ISBN: 0393316041.

Florijn, G., Meijers, M., van Winsen, P., 1997. **Tool Support for Object-Oriented Patterns**. In: **ECOOP '97: European Conference on Object-Oriented Programming**. Vol. 1241 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Heidelberg, pp. 472–495.

Fowler, M., 2002. **Patterns of Enterprise Application Architecture**. Addison Wesley Professional, Boston, MA, USA, ISBN: 0321127420.

France, R., Kim, D.-K., Ghosh, S., Song, E., 2004. **A UML-Based Pattern Specification Technique**. IEEE Transactions on Software Engineering 30 (3), 193–206.

Frederick, C., 2003. **Extreme Programming: Growing a Team Horizontally**. In: Marchesi, M., Succi, G. (Eds.), **XP/Agile Universe 2003**. Vol. 2573 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, pp. 9–17.

Gabriel, R., 1998. **The Failure of Pattern Languages**. In: Rising, L. (Ed.), **The Patterns Handbook**. Cambridge University Press, Melbourne, Australia, pp. 333–343, ISBN: 0-521-64818-1.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Boston, MA, USA, ISBN: 0201633612.

Gardner, K. M., Rush, A., Crist, M. K., Konitzer, R., Teegarden, B., 1998. **Cognitive Patterns**. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0-521-64998-6.

Gilb, T., 1985. **Evolutionary Delivery Versus the "Waterfall Model"**. ACM SIGSOFT Software Engineering Notes 10 (3), 49–61.

Guéhéneuc, Y.-G., Albin-Amiot, H., 2001. **Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects**. In: **TOOLS 39: 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001**. Santa Barbara, CA, USA, pp. 296–305.

Guimaraes, T., 1983. **Managing Application Program Maintenance Expenditures**. Communications of the ACM 26 (10), 739–746.

Hallsteinsen, S., Swane, E., 2002. **Handling the Diversity of Networked Devices by Means of a Product Family Approach**. In: **Software Product-Family Engineering**. Vol. 2290 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, pp. 264–281.

Haugland, S., 2003. **Dating Design Patterns**. Published by Solveig Haugland, ISBN: 0974312002.

Henry, E., Faller, B., 1995. **Large-Scale Industrial Reuse to Reduce Cost and Cycle Time**. IEEE Software 12 (5), 47–53.

Hosmer, D. W., Lemeshow, S., 2000. **Applied Logistic Regression**, 2nd Edition. John Wiley & Sons Inc., New York, USA, ISBN: 0471356328.

Huston, B., 2001. **The Effects of Design Pattern Application on Metric Scores**. Journal of Systems and Software 58 (3), 261–269.

Infragistics, 2003. **Expense Application—Reference Application**. URL `http://www.infragistics.com/products/thinreference.asp`

ISO, 1998. **ISO/IEC 10746: Information Technology—Open Distributed Processing – Reference Model**. URL `http://www.iso.org/iso/en/CombinedQueryResult.CombinedQueryResult?queryString=10746`

Jaccard, J., 2001. **Interaction Effects in Logistic Regression**. Quantitative applications in the social sciences. Sage Publications, Thousand Oaks, CA, USA, ISBN: 0761922075.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. **The Unified Software Development Process**. Addison-Wesley Professional, Boston, MA, USA, ISBN: 0201571692.

Kassem, N., 2000. **Designing Enterprise Applications with the J2EE Platform**. Addison-Wesley, Boston, MA, USA, ISBN: 0201702770.

Keller, R., Schauer, R., Robitaille, S., Pagé, P., 1999. **Pattern-Based Reverse-Engineering of Design Components**. In: **ICSE '99: 1999 International Conference on Software Engineering**. ACM Press, Los Angeles, CA, USA, pp. 226–235.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El-Emam, K., Rosenberg, J., 2002. **Preliminary Guidelines for Empirical Research in Software Engineering**. IEEE Transactions on Software Engineering 28 (8), 721–734.

Kleinbaum, D. G., 1994. **Logistic Regression : A Self-Learning Text**. Statistics in the Health Sciences. Springer-Verlag Heidelberg, New York, ISBN: 0-387-94142-8.

Kramer, C., Prechelt, L., 1996. **Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software**. In: **Third Working Conference on Reverse Engineering, 1996**. IEEE Computer Society, Monterey, CA, USA, pp. 208–215.

Larman, C., 2001. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process**, 2nd Edition. Prentice Hall, Upper Saddle river, NJ, USA, ISBN: 0130925691.

Liang, K., Zeger, S., 1986. **Longitudinal Data Analysis Using Generalized Linear Models**. Biometrika 73, 13–22.

Lientz, B. P., Swanson, E. B., Tompkins, G. E., 1978. **Characteristics of Application Software Maintenance**. Communications of the ACM 21 (6), 466–471.

Lindsay, R., Ehrenberg, A., 1993. **The Design of Replicated Studies**. The American Statistician 47 (3), 217–228.

McCullagh, P., Nelder, J., 1989. **Generalized Linear Models**. Chapman and Hall, New York, USA, ISBN: 0412317605.

McIlroy, D., 1968. **Mass Produced Software Components**. In: Naur, P., Randell, B., Buxton, J. (Eds.), **Software Engineering: Concepts and Techniques. NATO Conferences**. Petrocelli, Garmisch, Germany, pp. 138–156.

Microsoft, Inc, 2003a. **Application Architecture for .NET: Designing Applications & Services**. Microsoft Press, Redmond, WA, USA, ISBN: 0735618372.

Microsoft, Inc, 2003b. **Duwamish 7.0**. URL `http://msdn.microsoft.com/netframework/downloads/samples/?pull=/library/en-us/dnbda/html/bdasampduwam7.asp`

Microsoft, Inc, 2003c. **Microsoft .NET Pet Shop 2.0**. URL `http://msdn.microsoft.com/netframework/downloads/samples/?pull=/library/en-us/dnbda/html/bdasamppet.asp`

Microsoft, Inc, 2004. **C# Programmer's Reference: Foreach, in**. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrftheforeachstatement.asp`

MiniTAB, Inc, 2003. **MiniTab 13.32**. URL `http://www.minitab.com`

Neumann, G., Zdun, U., 2002. **Pattern-Based Design and Implementation of An XML and RDF Parser and Interpreter: A Case Study**. In: **ECOOP '02: 16th European Conference on Object-Oriented Programming**. Vol. 2374 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, University of Mlaga, Spain, pp. 392–414.

Ngu, A., 2003. **CS5369A Enterprise Application Integration**. URL `http://www.cs.swt.edu/ hn12/teaching/cs5369/2003Spring/admin/intro.html`

Öberg, R., 2003. **Review of "The Petstore Revisited: J2EE vs .NET Application Server Performance Benchmark"**. URL `http://www.google.com/search?q=cache:8OPCFEFDFd0J:www.dreambean.com/petstore.html+petstore+java+experience&hl=en&ie=UTF-8`

Object Management Group, 1995 1995. **CORBAServices: Common Object Services Specification**.

Object Management Group, 2004. **UML 2.0 Specifications**. URL http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

Oracle, Inc, 2003. **Oracle9iAS Containers for J2EE User's Guide Release 2 (9.0.2)**. URL http://otn.oracle.com/tech/java/oc4j/doc_library/902/A95880_01/html/toc.htm

Perforce, Inc, 2004. **Perforce Software Configuration Management System**. URL http://www.perforce.com/

Peters, L. J., Tripp, L. L., 1976. **Is Software Design Wicked?** Datamation 22 (5), 127–.

Prechelt, L., 2000. **An Empirical Study of Working Speed Differences Between Software Engineers for Various Kinds of Task**. Submitted to IEEE Transactions on Software Engineering, to be revised.

Prechelt, L., Unger, B., 1999. **Methodik und Ergebnisse einer Experimentreihe über Entwurfsmuster**. Informatik - Forschung und Entwicklung 14 (2), 74–82.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., Votta., L. G., 2001a. **A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions.** IEEE Transactions on Software Engineering 27 (12), 1134–1144.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., Votta., L. G., 2001b. **A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions.** IEEE Transactions on Software Engineering 27 (12), 1134–1144.

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W. F., 2002. **Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance**. IEEE Transactions on Software Engineering 28 (6), 595–606.

Rational, Inc, 2003. **PearlCircle Online Auction for J2EE**. URL http://www.rational.com/rda/wn_2002.jsp?SMSESSION=NO#pearlcircle

Reimer, D., Srinivasan, H., 2003. **Analyzing Exception Usage in Large Java Applications**. In: Romanovsky, A., Dony, C., Knudsen, J. L., Tripathi, A. (Eds.), **ECOOP '03: Workshop: Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms**. Darmstadt, Germany, pp. 10–19.

Rising, L., 1998. **The Patterns Handbook**. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0521648181.

Rising, L., Firesmith, D. G., 2001. **Design Patterns in Telecommunica-**

tions Software. Cambridge University Press, Cambridge, United Kingdom, ISBN: 0521790409.

Rittel, H. W. J., Webber, M. M., 1973. **Dilemmas in a General Theory of Planning**. Policy Sciences 4 (2), 155–169.

Rost, J., 2004. **Is "Factory Method" Really a Pattern?** ACM SIGSOFT Software Engineering Notes 29 (5), 1–1.

Schauer, R., Keller, R., 1998. **Pattern Visualization for Software Comprehension**. In: **IWPC '98: 6th International Workshop on Program Comprehension, 1998**. pp. 4–12.

Schmidt, D., 2002. **How to Hold a Writer's Workshop**. URL `http://www.cs.wustl.edu/ schmidt/writersworkshop.html`

Schmidt, D., Stephenson, P., 1995. **Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms**. In: **ECOOP '95: European Conference on Object-Oriented Programming**. Vol. 952 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Århus, Denmark, pp. 399–423.

Schmidt, D. C., 1994. **Reactor: An Object Behavioural Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching**. In: Coplien, J. O., Schmidt, D. C. (Eds.), **PLoP 94**. Addison-Wesley, pp. 529–545.

Scientific Toolworks Inc., 2003. **Understand for C++**. URL `http://www.scitools.com/`

SGI, 2004. **Standard Template Library Programmer's Guide**. URL `http://www.sgi.com/tech/stl/`

Singh, I., Stearns, B., Johnson, M., Team, E., 2002. **Designing Enterprise Applications with the J2EE Platform**. Addison-Wesley, Boston, MA, USA, ISBN: 0201787903.

Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E., Vokáč, M., 2002. **Conducting Realistic Experiments in Software Engineering**. In: **ISESE 2002: First International Symposium on Empirical Software Engineering**. IEEE Computer Society, Nara, Japan, pp. 17–26.

Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Vokáč, M., 2003. **Challenges and Recommendations When Increasing the Realism of Controlled Software Engineering Experiments**. In: Conradi, R., Wang, A. I. (Eds.), **ESERNET 2001-2002**. Vol. 2765 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, pp. 24–38.

Sjøberg, D. I. K., Kampenes, V. B., Hannay, J. E., Hansen, O., Karahasanovic,

A., Liborg, N.-K., Rekdal, A. C., 2004. **A Survey of Controlled Experiments in Software Engineering**. Submitted to IEEE Transactions on Software Engineering.

Smith, D., Robertson, W., Diggle, P., 1996. **Object-Oriented Software for the Analysis of Longitudinal Data in S**. Tech. Rep. Technical Report MA96/192, Department of Mathematics and Statistics, University of Lancaster.

Sun Microsystems, Inc, 2002. **J2EE Patterns Catalog**. URL `http://java.sun.com/blueprints/patterns/j2ee_patterns/index.html`

Sun Microsystems, Inc, 2003. **Java Pet Store Demo 1.1.2**. URL `http://java.sun.com/blueprints/code/jps11/docs/index.html`

Sun Microsystems, Inc, 2004. **Java API Documentation: Interface Collection**. URL `http://java.sun.com/j2se/1../docs/api/java/util/Collection.html`

TechExcel, 2004. **DevTrack Defect Tracking Tool**. URL `http://www.techexcel.com/products/devtrack/dtoverview.html`

The Hillside Group, 2004a. **Design Patterns Conferences**. URL `http://hillside.net/conferences/`

The Hillside Group, 2004b. **Shepherding**. URL `http://hillside.net/shepherding.html`

Thomas, W. T., Delis, A., Basili, V. R., 1995. **An Analysis of Errors in a Reuse-Oriented Development Environment**. Tech. Rep. CS-TR-3424, University of Maryland, Institute of Advanced Computer Studies, USA.

van der Linden, F., 2002. **Software Product Families in Europe: The Esaps & Café Projects**. IEEE Software 19 (4), 41–49.

van der Linden, F., Muller, J., 1995. **Composing Product Families from Reusable Components**. In: **1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, 1995**. IEEE Computer Society, Tucson, AZ, USA, pp. 35–40.

Vokáč, M., 2005a. **A Tool for Recovering Design Patterns from C++ Code, and its Application in a Case Study**. Journal of Object Technology, To appear July/August 2005.

Vokáč, M., 2005b. **Defect Frequency and Design Patterns: An Empirical Study of Industrial Code**. IEEE Transactions on Software Engineering Accepted for publication.

Vokáč, M., Jensen, O., 2004. **Using a Reference Application with Design Patterns to Produce Industrial Software**. In: Bomarius, F., Iida, H. (Eds.),

Product Focused Software Process Improvement. Vol. 3009 of Lecture Notes in Computer Science. Springer-Verlag Heidelberg, Kansai Science City, Japan, pp. 333–347.

Vokáč, M., Tichy, W., Sjøberg, D. I. K., Arisholm, E., Aldrin, M., 2004. **A Controlled Experiment Comparing the Maintainability of Programs Designed with and Without Design Patterns: A Replication in a Real Programming Environment**. Empirical Software Engineering 9 (3), 149–195.

Whitcomb, M., Clark, B., 1989. **Pragmatic Definition of An Object-Oriented Development Process for Ada**. In: **Tri-Ada '89: Ada Technology in Context: Application, Development, and Deployment**. ACM Press, Pittsburgh, Pennsylvania, United States, pp. 380–399.

Williams, R. D., 1975. **Managing the Development of Reliable Software**. In: **International Conference on Reliable Software**. ACM Press, Los Angeles, California, pp. 3–8.

Yacoub, S. M., Ammar, H. H., 2004. **Pattern Oriented Analysis and Design (POAD): Composing Patterns to Design Software Systems**. Addison Wesley, Boston, MA, USA, ISBN: 0201776405.

Yin, R., 2003. **Case Study Research, Design and Methods**, 3rd Edition. Sage Publications, Thousand Oaks, CA, USA, ISBN: 0-7619-2552-X.

Yourdon, E., 1976. **How to Manage Structured Programming**. Prentice Hall PTR, Indianapolis, Indiana, USA, ISBN: 0917072022.

Yourdon, E., 1999. **Death March**. Prentice Hall PTR, Indianapolis, Indiana, USA, ISBN: 0130146595.

Bibliographic index

This is an index of literature references. It is ordered alphabetically by author and lists the page or pages where this work is referred to in the text.

185, 198
Gardner *et al.* (1998), 9
Gilb (1985), 6
Guéhéneuc and Albin-Amiot (2001), 34, 140, 148, 181
Guimaraes (1983), 9
Hallsteinsen and Swane (2002), 112, 113
Haugland (2003), 9
Henry and Faller (1995), 120
Hosmer and Lemeshow (2000), 163
Huston (2001), 144
ISO (1998), 116
Infragistics (2003), 114
Jaccard (2001), 163
Jacobson *et al.* (1999), 4
Kassem (2000), 113
Keller *et al.* (1999), 34, 148, 181
Kitchenham *et al.* (2002), 28
Kleinbaum (1994), 155
Kramer and Prechelt (1996), 34, 148, 179
Larman (2001), 33, 124, 140
Lientz *et al.* (1978), 9
Lindsay and Ehrenberg (1993), 31, 32
McIlroy (1968), 113
Microsoft, Inc (2003a), 113
Microsoft, Inc (2003b), 114
Microsoft, Inc (2003c), 114
Microsoft, Inc (2004), 188
MiniTAB, Inc (2003), 153, 196
Neumann and Zdun (2002), 33, 142
Ngu (2003), 119
Object Management Group (1995), 186
Object Management Group (2004), 4
Oracle, Inc (2003), 122
Perforce, Inc (2004), 147
Peters and Tripp (1976), 4
Prechelt *et al.* (2001), 33, 140
Prechelt *et al.* (2002), 33
Prechelt and Unger (1999), 33, 140
Rational, Inc (2003), 114
Reimer and Srinivasan (2003), 119
Rising and Firesmith (2001), 9
Rising (1998), 33, 112, 140
Rittel and Webber (1973), 4
Rost (2004), 7
SGI (2004), 188
Schauer and Keller (1998), 34, 148, 181
Schmidt and Stephenson (1995), 33, 142
Schmidt (1994), 186
Schmidt (2002), 8
Scientific Toolworks Inc. (2003), 149, 190
Singh *et al.* (2002), 118
Sjøberg *et al.* (2002), 28, 31
Sjøberg *et al.* (2003), 28
Sjøberg *et al.* (2004), 30, 31
Sun Microsystems, Inc (2002), 113, 115
Sun Microsystems, Inc (2003), 11, 112, 114, 119
Sun Microsystems, Inc (2004), 188
TechExcel (2004), 147
The Hillside Group (2004a), 8
The Hillside Group (2004b), 8
Thomas *et al.* (1995), 113
Vokáč *et al.* (2004), 10, 11, 30, 31, 34, 141
Vokáč and Jensen (2004), 10, 35
Vokáč (2005), 148
Vokáč (2005a), 10, 34, 35
Vokáč (2005b), 10, 34, 35
Whitcomb and Clark (1989), 6
Williams (1975), 6
Yacoub and Ammar (2004), 8
Yin (2003), 27, 116