Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project

Peter Wendorff
ASSET GmbH, 46147 Oberhausen, Germany
ASSET_GmbH@t-online.de

Abstract

Design patterns have been eagerly adopted by software developers in recent years. There is ample evidence that patterns can have a beneficial impact on software quality, but in some cases patterns have been inappropriately applied due to a lack of experience.

This paper reports on a large commercial project where the uncontrolled use of patterns has contributed to severe maintenance problems. As a result a substantial reengineering effort was undertaken, that led to the identification of a number of inappropriately applied patterns. At first glance the elimination of these patterns appears to be desirable, but often they are tightly coupled to other software artefacts, so that their removal is economically not viable.

1. Introduction

Since their inception in the early 90s design patterns (henceforth simply called "patterns") have been adopted enthusiastically by the software community, both scientists and practitioners. Their acceptance by practitioners has clearly been helped by the intuitively appealing ideas behind patterns. There is a multitude of success reports from commercial organisations, a typical example of this is given in [1].

As patterns have become popular many software developers have applied patterns in an exploratory way. Much of that code has entered the maintenance phase recently. The issue of reengineering code containing questionable patterns will therefore gain importance in the near future. This observation has motivated this paper.

We draw on our practical experience gained during a large commercial project that has taken place at one of Europe's largest service companies between 1994 and 2000. During this period several operational versions of the software were produced and rolled out, due to functional enhancements and changes in technology. The software is completed now and has been promoted to the maintenance phase recently. The project in question caused an effort of several hundred man-years, with more than 50 programmers involved at peak times. The code size is 1000 KLOC (800 KLOC C++ and 200 KLOC PL/SQL (with 1 KLOC = 1000 lines of code)). This paper is written by a software engineer with about 8 years of professional experience who was assigned to the project for 8 months during a major reengineering effort as an external consultant.

Because the project was of high strategic importance a number of external experts from leading software consultancies were hired throughout the project. Much attention was paid to technological issues, in particular aspects relating to software architecture. This contributed to an atmosphere where the use of patterns was encouraged.

In this paper we adopt a reengineering point of view. Therefore we concentrate on patterns actually found in production code. We do not discuss how to improve the use of patterns in future, we rather focus on remedies for past inappropriate use of patterns.

At present the subject of patterns is only scarcely covered in publications from a reengineering perspective. The article [4] is a report about the reengineering of object-oriented software in the telecom industry, and the author notes that the inappropriate application of object-oriented techniques in the past has created a new class of legacy systems. The author gives some room to "pattern restructuring" as a possible reengineering approach, but the article does not pay particular attention to patterns. The book [6] is an excellent work on reengineering of code, but it does barely cover the inappropriate use of patterns, let alone remedies for that problem. The paper [5] is a very interesting and ironic account of inappropriate use of patterns by software engineers, but the paper does not provide practical advice on how to

0-7695-1028-0/01 $10.00 © 2001 IEEE

address the problem. In [11] the authors report the results of a controlled experiment. Their valuable empirical research investigates the effects of design patterns on software maintenance effort, but they do not discuss procedures to deal with inappropriate patterns.

In section 2 we discuss some theoretical issues related to patterns. In section 3 we present typical problems that can be introduced into source code by the unreflective use of patterns. In section 4 we will present a sequence of simple steps for the assessment of patterns from a reengineering perspective.

2. Aspects of Patterns

In this section we will discuss some aspects of patterns that are relevant in the context of this paper, but naturally we cannot provide a comprehensive treatment of the subject.

2.1. Basic Elements

In [7] the following definition is given: "The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". The essential elements of a pattern are [7]:

- the "pattern name" as denotation of the concept
- the "problem" as description of the class of situations when the concept is applicable
- the "solution" which gives an abstract description of a generic system of elements and their relationships that solves the problem
- the "consequences" that result from the application of the solution, for example trade-offs, costs, and benefits

2.2. Critical Success Factors

There exist a lot of critical success factors (CSFs) for the application of a pattern, and empirical research shows that these factors are barely understood at the present state of the art [11]. Some CSFs are discussed in books like [3] or [7] in an informal way, leaving much space for personal opinion. Therefore the decision to apply a pattern in a particular situation is to a large degree a matter of subjective judgement. Popular collections of patterns like [9], which are frequently consulted by programmers, do barely discuss critical success factors at all.

2.3. Design Options

A pattern is not necessarily a best practice, it is rather a proven practitioner's approach. Therefore a pattern is merely one design option among others during software development. The decision to favour one of these options over the others should normally be based on the detailed and specific quality goals of the software project. Therefore the choice in favour or against the use of a pattern must be based on the quality goals of the project.

2.4. Flexibility

One key concern of patterns is flexibility. In [7] Gamma et al. note: "A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, client modification, and retesting. Redesign affects many parts of the software system, and unanticipated changes are invariably expensive. Design patterns help you avoid this by ensuring that a system can change in specific ways". The crucial point to notice here is that a pattern will usually add to a particular aspect of flexibility, but it cannot provide universal flexibility. If a pattern is applied to enhance flexibility of a software design, careful judgement is required to ensure that the pattern promotes the desired aspect of flexibility.

2.5. Cost

Gamma et al. note that the flexibility introduced by patterns comes at a price when they discuss the mechanism of delegation that is used by many patterns: "Delegation has a disadvantage it shares with other techniques that make software more flexible through object composition: Dynamic, highly parameterized software is harder to understand than more static software. There are also run-time inefficiencies, but the human inefficiencies are more important in the long run".

Patterns usually lead to an increased number of software artefacts (e.g. classes, files, associations, etc.), which normally increase the static complexity of a software system considerably. Furthermore, when the additional associations are instantiated at run-time, they result in additional and often volatile links between objects, and this usually increases the dynamic complexity of a software system significantly.

2.6. Cost of Change

Recently "Extreme Programming (XP)" has been proposed as a new "lightweight" software engineering methodology [2], and it is immensely popular among practitioners. XP is based on four values, one of which is "simplicity". This idea is explained in [2]: "XP is making

a bet. It is betting that it is better to do a simple thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway".

We will not give our position on XP here, we rather note that XP is a methodology that has emerged in practice and represents lessons learned by large numbers of experienced software engineers. Clearly there is a case for simplicity from an ecomomic point of view. As the application of patterns frequently leads to more complex software, there may be reasons to caution their use.

2.7. Software Quality

The term "software quality" already denotes an elusive and multidimensional concept [8]. None of the leading textbooks [3], [7], or [9] on patterns refers to any explicit model of software quality, but their authors frequently mention software quality attributes like "flexibility", "comprehensibility", "maintainability", etc. without giving clear operational definitions for these concepts.

It is a typical situation that different software quality attributes are in a conflicting relationship to another [8]. For example it is possible that the application of a pattern results in code that is more flexible, but that is more complex as well. Whether the overall economic effect of a pattern is positive or negative in such a case is therefore dependent on future needs that are not fully anticipated at the time of design and coding. It would be a breakthrough in software engineering, if we could predict software quality attributes in a comprehensive and objective way. Sadly we are far from that situation [10].

Therefore the decision in favour or against the use of a pattern is to some degree a matter of subjective judgement for at least two reasons. First, our ability to measure the effects that patterns have on software quality are very limited [10]. Second, there is no firm and formal theory that links patterns to software quality concepts, and therefore we do not have an explicit model of how patterns work, what their critical success factors are, how these factors interact, etc. [11].

2.8. Software Lifecycle

Any planning situation that has to deal with a dynamic environment must care about uncertainty. A constructive way to deal with this uncertainty is to build flexibility into a product, so that it can be modified easily. Patterns provide a way to build flexibility into a software product.

In the course of a software development project the degree of uncertainty often decreases considerably. For example as experience in the application domain increases, the number of relevant environmental factors can often be narrowed down. If uncertainty decreases, then the need for flexibility may decrease accordingly. Therefore it is conceivable that a pattern that is introduced into a design at an early stage of the software development process may become obsolete at a later stage.

2.9. Removal

In the case of the removal of a pattern from code we have to balance three conflicting issues (cf. [12]): First the possible benefit of the pattern. Second the possible extra cost of the pattern. Third the cost of its removal.

2.10. Empirical Research

The patterns movement was not theory-driven at its inception, and it still is not at the time of writing, it is rather an eclectic practitioner's approach. This is reflected by the lack of empirical research into patterns. The severe lack of theoretical groundwork makes it difficult to assess the cost/benefit ratio of patterns objectively. In [11] the authors elaborate the blurred operational definitions of software quality concepts claimed in favour of patterns. Their work shows that contrary to common belief the beneficial effects of patterns are not universally obvious. Therefore it must be concluded that at the present state of affairs there is a serious lack of hard scientific evidence for the usefulness of patterns in many situations.

3. Inadequate Application of Patterns

In this section we will give some examples from our reengineering work where the application of patterns has been questionable. This is not intended to criticise these widely accepted patterns. Instead we want to illustrate how and why patterns were in some cases improperly applied by software engineers.

3.1. The Proxy Pattern

The "proxy" pattern as described in [3], [7], and [9] is a simple one with an easy to grasp rationale behind it. Its simplicity makes it a typical "beginner's pattern", and according to our experience, beginners tend to use it freely.

The basic idea behind the proxy pattern is to use a placeholder (called the "proxy") for another object (called the "real subject") in order to control access to the real subject [7]. The proxy constitutes an additional layer of indirection and manages the requests made by clients to

the real subject. The proxy and its real subject exhibit the same interface, and therefore the proxy can be substituted wherever its corresponding real subject is expected by clients. A typical application of the proxy pattern are expensive operations, where the proxy decides when and how to forward the requests to the real subject, in order to ensure efficient processing.

During our reengineering analysis the value of many proxies found in the code seemed to be doubtful. We noticed the following problems:

- In many cases developers justified the application of the proxy pattern with expected future needs for flexibility, access control, and performance. In fact in most cases these future needs never materialised. In these cases the proxy pattern often remained in the code, the rationale behind its use disappeared.
- The Proxy pattern naturally leads to a substantial increase in the number of classes (and usually files). For example the simple variant of the proxy patterns described in [7] necessitates one interface class and two concrete classes, compared to a single class for a simple solution. This means that the size and complexity of the software increases considerably.
- In [3] it is proposed to include some kind of pre-processing and post-processing for a request to a class in the proxy class of that class. Surely this division of responsibilities between classes makes sense in some special cases, but it can make the interaction between objects pretty complicated. We found a case where considerable and complicated functionality was scattered over a hierarchy of proxy classes without a documented or conceivable rationale.
- The proxy pattern introduces an additional level of indirection which impedes comprehension of the dynamic flow of control at run-time because it bloats the call stack. This can make debugging much harder.

During our reengineering analysis we put every proxy pattern used in the code to the test and removed a large number of proxies completely.

In many cases the proxy objects did simply forward requests to their real subjects directly without adding any processing. As is correctly noted in [9] these proxies in their "pure form" are not useful. Fortunately their removal is straightforward, because one can simply substitute the real subject for the proxy. This still is cumbersome and monotone work, but the economies of classes and files are usually substantial. Furthermore this work can be done in a mechanical fashion and does not involve the risk of introducing errors into the code.

The removal of a proxy pattern from the code was straightforward in cases where there was a clear separation of responsibilities between the proxy class and the other classes involved in the pattern.

The removal of a proxy pattern in presence of complex pre-processing or post-processing, proved to be very difficult and needed careful attention to side effects.

In one subsystem with about 3000 lines of code we removed 3 out of 7 proxy patterns altogether, leading to a reduction of 200 lines of code.

3.2. The Observer Pattern

The observer pattern as described in [3], [7], and [9] is a well-known pattern with a long history. Originally it emerged as part of the Smalltalk environment where it had been developed to facilitate GUI design, and where it is known as model-view-controller (MVC) pattern. Since those early days the concept has been refined and generalised, and it has been applied to other domains as well.

The original MVC approach divides an interactive graphical application into two fundamental parts: an abstract application (the "model") and a user interface (the "views" and the "controllers") that handle all I/O functions. The MVC defines a generic protocol for the communication of these three kinds of parts. There are two important aspects of the MVC approach. First, it is particularly suited in the case of a one-to-many dependency between a model and several views [7]. Second it facilitates reuse of software through the generic protocol and the clear division of labour between the participating objects.

In our project a framework for GUI implementation (an industry standard) was used that provides a comprehensive and reasonable support infrastructure for modal dialogues. To our utter surprise one programmer had not used this proven and free architecture but rather implemented a complex MVC model on his own. From an economic point of view this was clearly an extremely bad solution for three reasons. First, by its very nature a modal dialogue does not have multiple views. Second, in our project it was never considered to reuse the dialogues in any way. Third, the modal dialogues in question were rather primitive and did simply not necessitate the separation of concerns promoted by the MVC model.

A post mortem analysis of the decision process that led to the uneconomical decision in favour of the MVC approach revealed that the programmer in question

- wanted to gain experience with the MVC approach
- justified his design decision in terms of "flexibility" and "reusability"
- found it simple to convince the young and inexperienced project leader by referring to the purported superiority of patterns over all other solutions

An important aspect in this case is the use of the buzzwords "flexibility" and "reusability". These two buzzwords are ascribed to certain patterns in most of the

literature on patterns (cf. [3], [7], and [9]). Clearly these two quality attributes generally do not harm, but in this case they were not required and they were achieved at an irresponsible cost.

From an economic point of view it would have been desirable to remove the MVC approach from the code. If the aforementioned industry standard had been used, that would have resulted in a fraction of the code size that was effected by the MVC approach.

Unfortunately the removal of the MVC architecture would have amounted to a completely new design and implementation of the graphical user interface and was therefore out of the question.

3.3. The Bridge Pattern

The bridge pattern [7], [9] provides a way to decouple an abstraction and its implementation. There is a number of potential situations when full independence between abstractions and their implementations makes sense, but these situations are rather special in nature.

Generally abstraction is not a value in itself, instead it is a means to support human understanding and communication. Therefore a software design should exhibit a useful degree of abstraction, ideally that one with the best cost/benefit ratio in a given situation.

Many programming languages provide language features that allow a certain degree of independence between an abstraction and its implementation. For example in our project C++ has been used, and this language makes a distinction between a class declaration and its corresponding implementation, and the two are usually placed in separate files. We think that one should first carefully evaluate the abstraction mechanisms directly supported by the implementation language. Only if there is evidence that these abstraction mechanisms are not sufficient in the given context, then more complicated and indirect abstraction mechanisms should be considered.

The case where there is only one implementation for an abstract concept is mentioned as "a degenerate case of the Bridge pattern" in [7], and the authors note that this pattern has very limited applicability. We found degenerate bridge patterns several times during our reengineering work. Seemingly some of the programmers in the project were beset by the idea of separating an abstraction and its single corresponding implementation.

The post mortem examination of the design decisions in favour of the bridge pattern revealed a recurring course of events: During the early design stages the designers frequently overestimated the need for later extensions of the software. Faced with uncertain future requirements they usually opted for the more flexible solution. In many cases the bridge pattern was chosen in a rather mechanical fashion.

Later many of the premeditated change requests did not materialise, and in these cases the bridge pattern usually stayed in place even though the rationale behind its use had disappeared completely.

It is noteworthy that in several cases the original rationale behind the choice of the bridge pattern could not be established at all, due to a lack of appropriate documentation. This does compromise the idea of flexibility supported by patterns. It is usually helpful to know the kind of flexibility that is provided by a pattern in a particular context, in order to exploit this inherent flexibility efficiently. If the corresponding documentation is not present, then relevant options provided by the pattern may go unnoticed.

During our reengineering work we only addressed the degenerate form of the bridge pattern. Removing such a bridge pattern from code is usually a straightforward merger of two classes into one class. In one subsystem we removed 2 out of 3 bridge patterns with an economy of 190 out of 1400 lines of code.

3.4. The Command Pattern

The command pattern [7], [9] (cf. the more elaborate pattern "command processor" in [3]) is based on the representation of commands by objects. Thereby complex command structures can be represented by a flexible object model. This approach can be used to design applications that are easily extensible.

In our project the requirement was that about 100 elementary operations had to be performed in a number of well-defined sequences on a set of data. The original requirement was that the sequences should be read from a database. Therefore the designers decided to use the command pattern, which was a reasonable choice at that time.

Our post mortem examination of the ensuing design process told an interesting story. After the decision in favour of the command pattern the designers got inspired by the additional options associated with that pattern. A number of additional functions were added to the design that had never been mentioned in the requirements of the project. As a matter of fact these functions were added because they were noted in the popular text [7] as possible options, although they were not part of the requirements specification.

Then the requirements changed. The idea to read the sequences of operations from a database was dropped. At that stage the command pattern achitecture had developed into a very complex software artefact with loads of unnecessary and complicated features that was intricately

intertwined with other parts of the software achitecture. Even the original designers themselves freely admitted that their solution was completely over the top.

In this situation a full reengineering of the failed command pattern architecture would have been desirable, but because of its complexity that was no longer a viable option.

Clearly it would be unfair to blame patterns for the ill-judgement of individual software engineers. Nevertheless our investigations indicated that the cookbook style of much of the patterns literature might have stimulated the playfulness of some software engineers in the case of our project.

4. Removal of Patterns

During our work we have developed a simple procedure to guide the removal of inappropriate patterns from code. This is clearly an eclectic practitioner's approach that has evolved and succeeded during our project, but it is no comprehensive and systematic solution to the problem. We found the following steps helpful (cf. [12]):

Step 1: Identify the relevant quality attributes for the software under consideration.
Step 2: Identify patterns used in the code.
Step 3: Try to reconstruct the original rationale behind the use of patterns.
Step 4: Assess the concrete benefit of a pattern.
Step 5: Assess the concrete extra cost of a pattern.
Step 6: Assess the total effort needed to remove the pattern.
Step 7: Make a balanced decision based on Steps 4, 5, and 6 whether to remove the pattern.

To Step 1: Usually the detailed information system strategy of the company or project should provide some hints on the quality strategy. In our project the official plans for the system do not envisage substantial functional enhancements for the future. Therefore issues like extensibility and flexibility are minor issues. On the other hand the product must be maintained by a small team of maintenance engineers that were not involved in the development phase, and therefore low complexity of the code is a major concern to ensure maintainability.

To Step 2: Identifying patterns in the code turned out to be cumbersome at times. Often combinations of patterns had been used, which often led to a blurred separation of concepts. The same applies to variants of patterns, that were often difficult to tell. On top of that come simple errors in the application of patterns that impede their identification: For example we found patterns documented as "proxy" patterns, which according to [7] are "adaptor" patterns.

To Step 3: Reconstructing the original motivation behind the application of a pattern is an interesting, yet difficult activity. In our project much of the design documents originated in the mid 90s and were badly maintained or even outdated. At one stage one of the major software designers, who had left the project in the meantime, was hired for one day to discuss these issues. On the backcloth of the project's history this designer confirmed a number of our conjectures concerning inappropriately applied patterns.

During our post mortem analysis into the inappropriate use of patterns two general categories became apparent. In the first category are patterns that were simply misused and where the developers who used them had clearly not understood the rationale behind the pattern. In the second category are patterns that do not fall into the first category, but which did not match the project's set of quality criteria at the time of reviewing.

To Step 4: It is important to carry out this assessment in the light of the quality criteria (Step 1). For example, enhanced flexibility of a design due to the use of a pattern is only beneficial if flexibility is a relevant quality criterion. Moreover one should clarify the underlying idea of flexibility: Is the particular form of flexibility offered by the pattern really the desired one? Another important aspect of flexibility is that it is a potential benefit, i.e. it becomes beneficial only if it is actually used. Therefore the likelihood that a flexibility potential is really used one day should be included in the assessment. If this likelihood is close to zero, then the benefit of the pattern is negligible as well.

To Step 5: Again it is important to carry out this assessment in the light of the quality criteria (Step 1). We used a very simple approach based on two ratings. The first rating was the subjective degree of additional complexity introduced by a pattern, which was rated on an ordinal scale (low, medium, high). The second rating assessed the relevance for software developers to understand the bit of code containing the pattern, again rated on an ordinal scale (low, medium, high). The resulting assessment matrix gives a clear overview of the patterns with reference to possible removal. The idea behind the assessment matrix is that patterns with a high degree of additional complexity that affect code that must be understood by many software developers are clear candidates for removal.

To Step 6: A major determinant of the effort needed to remove a pattern from code is the degree of coupling it exhibits to other parts of the system. We only regarded one simple aspect of coupling, namely the number of files affected by the removal of a pattern, i.e. the number of

files of which a new version would be checked into the configuration management system as a result of the pattern's removal. Admittedly this simplistic measure for the software quality attribute coupling is very coarse, and it was primarily chosen for pragmatic reasons, because it can be calculated using the dependency checking mechanism of an ordinary compiler. Unfortunately we did not have more sophisticated tools at our disposal.

To **Step 7**: The decision to include a pattern in program code is to some extend a subjective one, and naturally the same applies to the removal of a pattern. Nevertheless we have found one rule of thumb very helpful: Don't give a pattern the benefit of the doubt.

5. Conclusions

The intention of this paper is not to criticise the idea of patterns, instead it is motivated by the observation that the inappropriate application of patterns can possibly backfire. We have presented several examples where proven and popular patterns have been applied by professional designers and programmers, probably with the best of intentions. These examples show that this has led to negative results in some cases. Therefore we believe that inappropriately applied patterns will be a relevant aspect of software reengineering and maintenance in the future.

The benefits of patterns are eagerly acknowledged in the literature, but the associated costs feature less prominently. Naturally patterns affect different software quality attributes, and the application of a pattern may for example result in a desirable increase of flexibility at the cost of an undesirable increase of complexity. Therefore the use of patterns is not a "free lunch". Accordingly it can make economic sense to remove an inappropriate pattern from code.

We have found two categories of inappropriately applied patterns in our project. In the first category are patterns that were simply misused by software developers who had not understood the rationale behind the patterns. In the second category are patterns that do not fall into the first category, but which do not match the project's requirements.

Our analysis of patterns in the second category has identified a number of situations that gave rise to inappropriate patterns. First, many software developers routinely overestimated the future volatility of requirements, and often opted for patterns to build flexibility into the software. Second, requirements changed over the lifetime of the project, and thereby patterns that had been reasonable at first became obsolete later. Third, in some cases patterns were applied without any regard to the quality goals of the project, for example because software developers wanted to gain experience with patterns. Fourth, the cookbook style of much of the patterns literature tempted to embellish a pattern with additional features. Therefore it happend that features were added because they were mentioned in books, not because they were actually needed.

One major impediment to our work was the lack of appropriate documentation of early design decisions in favour of patterns. We believe that software developers who fail to record these crucial decisions have simply not understood the informing ideas behind patterns.

We have presented a simple procedure consisting of seven steps that we have used during the identification, assessment, and removal of obsolete patterns. This procedure has led to more objective, well-documented, and economically sound decisions during our reengineering activies. Nevertheless the decision to remove a pattern inevitably remains subjective to some degree.

In the course of our work we identified several cases of inappropriate patterns where their removal would not have been viable. Often these patterns are strongly coupled to other software artefacts, giving rise to complex dependencies and possible side effects.

This paper is drawn on the experience gained in a single project, albeit a very large one. We believe that the situation described in this paper may apply to a considerable proportion of the software developed during the 90s, the heydays of patterns. If that is true, then we have to be prepared to deal with inappropriately applied patterns in the future. This leads the way for future research into the subject from a reengineering perspective. First, we need a more objective assessment of the cost/benefit ratio of patterns. Second, we need more guidance in the decision when to remove a pattern from code. Third, we should develop systematic procedures and tool support for the removal of inappropriate patterns from code.

6. References

[1] Beck, K. et al., "Industrial Experience with Design Patterns", Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, 1996, pp. 103-114.

[2] Beck, K., Extreme Programming Explained: Embrace Change, Addison-Wesley, 2000.

[3] Buschmann, F. et al., Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996.

[4] Casais, E., "Re-Engineering Object-Oriented Legacy Systems", The Journal of Object-Oriented Programming, Vol. 10, No. 8, 1998, pp. 45-52.

[5] Dodani, M., "Rules Are for Fools, Patterns Are for Cool Fools", The Journal of Object-Oriented Programming, Vol. 12, No. 6, 1999, pp. 21-23, p. 70.

[6] Fowler, M. et al., Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman, 1999.

[7] Gamma, E. et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

[8] Gillies, A., Software Quality: Theory and Management, International Thomson Computer Press, 1997.

[9] Grand, M., Patterns in Java (Vol. 1), Wiley & Sons, 1998.

[10] Pfleeger, S. L. et al., "Status Report on Software Measurement", IEEE Software, March/April 1997, pp. 33-43.

[11] Prechelt, L. et al., "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions", To appear in IEEE Transactions on Software Engineering, http://wwwipd.ira.uka.de/~prechelt/Biblio/.

[12] Pressman, R. S., Software Engineering - A Practitioner's Approach, McGraw-Hill Companies, 1997.