

Strategy Based Elimination of Design Flaws in Object–Oriented Systems

Adrian Trifu¹ and Iulian Dragos²

¹ FZI Forschungszentrum Informatik
Program Structures (PROST)
Haid-u-Neu-Str. 10-14, 76131 Karlsruhe, Germany
`trifu@fzi.de`

² "Politehnica" University
Faculty of Computer Engineering
B-dul V. Pârvan 2, 1900 Timișoara, România
`dragos@fzi.de`

Abstract. Due to frequent requirements changes, extensions of functionality, as well as continuous adaptation to changing environments, the designs of large software systems continually degrade. In the context of object–oriented software restructuring, we show that there is still an important gap between the design flaw detection and correction phases, as well as an acute lack of rigorously defined correction methodologies and appropriate tools. In this paper, we state a number of important criteria that have to be fulfilled by any such approach. The novel notion of “correction strategy” is then introduced, as a reference description that enables a human–assisted tool to plan all necessary steps for the safe removal of detected design flaws, with special concern towards the targeted quality goals of the restructuring process. Based on correction strategies, we propose a scalable, quality–driven and highly automatable approach to bridge the gap between state of the art design flaw detection methodologies and source code transformations.

1 Introduction

One of the problems with software in general is that not all requirements can be completely foreseen as the system is designed. During the lifetime of a software system, new requirements are continually formulated while existing ones are changed or even dropped. Due to this and to the continuous adaptation of the system to ever changing business environments, modifications and extensions of functionality are inevitable. As a consequence, the original design slowly but surely degrades, significantly increasing overall maintenance costs. This phenomenon has been known in literature as the software entropy effect, design drift [1], or software decay [2].

Rewriting a large decaying software system from scratch is too expensive so the solution is usually reengineering (also called software renovation) [3]. In most cases, the actual extension or adaptation of functionality is preceded by a *restructuring phase*, in which the structure of the system (the “how”) is changed without affecting the behavior (the “what”), so that a predefined set of quality factors are favored. In the case of functionality extension for example, such quality factors could be *extensibility*, *flexibility* and *performance*. By improving these aspects of “quality”, the system is better prepared to receive new functionality, and the

process of integrating this new functionality runs smoothly. In other words, *software restructuring can be viewed as a goal-directed process*. This means that good methodologies for software restructuring have to have a global scope and every phase, from the detection to the elimination of design flaws, has to be driven by these initial goals. We will refer to these goals as the *targeted quality factors* of the process. Furthermore, methodologies must be scalable, both to large systems, and to various levels of abstraction of design flaws. Finally, they must support heavy automation to make them practicable.

Unfortunately the current state of affairs is completely different. First, there is no clear understanding of exactly what a design flaw is. Often, detection techniques highlight the effects (symptoms) of design flaws rather than the real problem. Of course, what needs to be eliminated is the flaw and not the symptoms that it causes. Furthermore, there is still a large gap between detection and correction technology. In some existing approaches, the engineer is presented with a list of refactorings that are applicable to a system, but the choice of which one to apply is left to his own intuition. Moreover, we still lack a nuanced view over different levels of problem and solution abstraction as well as complexity, formal ways to describe the mapping between such com-

plex problems and solutions (we call these strategies), and predictive models that allow choosing the right solution to a given problem, with regard to the previously established quality targets. Although tool support has developed considerably in the last years, there are still no tools that support problem correction, while at the same time offering satisfactory control over the entire restructuring process.

To bridge the gap that exists between object-oriented problem detection and source code transformations, we propose a new approach, based on the novel concept of *correction strategies*. Correction strategies are human-made descriptions that outline possible ways of action concerning the removal of design flaws from object-oriented systems. Our vision is one in which complex flaws in object oriented design are precisely detected and eliminated, under the supervision of the software engineer, much in the same way as complex illnesses in a human patient are diagnosed and suitably treated, under the supervision of a doctor, in a predictive manner and with regard to the patient's medical history and overall state of health (in our case, the quality of the system).

The rest of the paper is structured as follows: First, we present the motivation and objectives of our work in section 2. In section 3 we propose our approach and illustrate it on a typical example. Section 4 is dedicated to an overview and critique of related work. Finally, section 5 draws conclusions and provides insight into our future work.

2 Motivation and Objectives

Something is still missing from today's state of the art concerning design flaw elimination from object-oriented systems, and that is a bridge that links state of the art *problem detection methodologies* with existing *correction techniques* (source code transformations). The software engineer is currently forced to quasi-manually and separately analyze each "suspect" code or design fragment, and based solely on his own experience and intuition, choose the appropriate combination of corrective measures to be applied, in such a way that it leads to a better design (improved target quality factors). Some help in this process is provided by *bad smells* [2], some of the *antipatterns* [4], as well as *reengineering patterns* [5], however, they are informal descriptions (therefore not automatable) and do not model the impact on quality in any way.

Throughout this paper, we will make an analogy with the medical field. In this analogy, the patient is our system. This patient undergoes a set of investigations with the purpose of diagnosing his illnesses.

We will use and distinguish between the terms "design illness" which denotes higher level problems in the design of a system, that have a direct negative influence on quality factors such as maintainability (e.g. "God Class" [6]), as opposed to the "symptoms" of that illness, which are usually abnormal measurement values given by certain software metrics (e.g. relatively large class size, low class cohesion, high coupling with many lightweight classes, etc). In the medical field, as in any field, a precise diagnostic of the responsible illness (flaw) is essential for an efficient treatment. Although there are very few, and only relatively recent, design flaw detection techniques today that attain the necessary level of abstraction, our approach will not deal with the detection phase, but instead rely on existing approaches (two examples are [7], [8]).

The lack of proper means of diagnostic has also hindered the development of satisfactory correction tools. The reengineering process is still a mostly manual, expensive and error prone activity.

Our objective is to bridge this gap with a two step approach: first, elaborate a formal mapping from illnesses to solutions (we call it *correction strategy*), and second, develop a methodology that with the help of our strategies, will lead to the successful elimination of detected flaws in object-oriented designs. In order to be of real use in reengineering large object-oriented systems, our approach (as indeed any approach) has to fulfill the following set of criteria:

- C1 high level of abstraction:** the approach should be language-independent, by operating on the level of design elements rather than source code;
- C2 quality driven:** always take into account the impact of (potential) transformations on the overall quality of the reengineered system and choose the optimal (or close to optimal) sequence of transformations. This is especially important since in many cases, there can be more than one "good" solution to a problem;
- C3 causality:** have the flaw as a starting point and look for the optimal solution. Some approaches start from a set of refactorings and try to see which one improves the system the most. This approach is in our opinion not suitable. Like in the medical field, drugs are not tried out on a patient to see which one produces the best result. Instead, an illness is first diagnosed and appropriate measures are taken as a direct consequence;
- C4 scalability:** the approach should scale from two standpoints: the size of the system and the abstraction level of design flaws. The second aspect means that a good methodology should be just

as effective at solving low level (e.g. intra-class) and high-level (e.g. involving several classes) design problems;

- C5 behavior preservation:** in order for the methodology to be used with confidence, all transformations involved should be shown to preserve the behavior of the system. While a purely formal demonstration is not possible, there has to be at least a semi-formal proof for non-pathological cases. This can be achieved with precondition and postcondition logic.
- C6 extensible:** the approach should be easily extensible to cover new design flaws;
- C7 automation:** the methodology should allow substantial levels of automation and tool support, an essential requirement for reengineering large systems. We do not aim to achieve full automation, however the role played by the human engineer should be as close to supervision as possible.
- C8 dissemination:** the developed methodology, techniques, set of common practices and tools should provide a means of recording and transmitting expertise on reengineering object-oriented systems, just as design patterns are a means of recording and transmitting expertise on object-oriented design.

3 Approach

Let us consider the example of an often-encountered design flaw: *the “switch” problem*. It appears as large conditional structures in code, over type or equivalent information, typically involving large “switch” statements on member variables or their types. This flaw has several related forms that have similar structure but different semantics, such as “lack of state” or “type checks in clients” [8, 1, 5]. It is schematically presented in figure 1.a. Depending on the semantics of the main actors, the flaw can be solved in several different ways, each having several variations. Two possible solutions are *simple subclassing* and the *state* design pattern, presented in figure 2 a and b.

Our approach for eliminating design flaws is based on the concept of *correction strategy*, which is a description of the solution(s) and their variations to a given flaw, together with some quality-related information. This information, together with an appropriate quality model and methodology, allows an automated tool (or very determined engineer) to estimate the impact of various solutions on overall system quality and consequently choose the best one. Please note that a correction strategy does not represent a “program”, or the complete algorithm to eliminate a design flaw. A methodology will specify

the way in which all major decisions are to be made. A future tool will implement this methodology and will use the information given in the strategy as a guide to eliminate a given flaw.

In order to be easier to understand, we will present the concept of correction strategies with the help of an extended BNF ([9]) grammar description. Although there is currently no concrete implementation of such a language, we believe that it provides a way to describe correction strategies in a natural, easy to grasp way.

In figure 3, some of the top nonterminals of the grammar are given, along with a schematic representation of an arbitrary strategy. As suggested in the figure, a strategy contains an interface section, followed by a combination of *sequences* and/or *nodes*. The *interface* section is meant as the mapping between actors in a detected design flaw and the correction strategy. Sequences (represented as curves in the figure) are defined as complex, behavior-preserving and atomic code transformations. They can contain refactorings, non behavior-preserving transformations, conditional and iterative constructs. The only requirement is that the sequence as a whole, must guarantee preservation of behavior. That is, if a specified set of preconditions are fulfilled, certain postconditions are true after executing the sequence, and external behavior is unchanged. In the strategy listed in figure 1.b, lines 22 and 36 represent calls to sequences. Portions of a sequence are listed in figure 4.

Nodes represent major decisions, that can impact system quality, such as the one of choosing between the state pattern or simple subclassing as the next step in a strategy. In figure 3, nodes are represented as filled circles, labelled *N1* through *N3*. Each node has at least two *branches*.

Branches are alternative paths that may in turn contain *sequences* and/or *nodes*. Node *N2* for example contains branches *B3* and *B4*, which in turn contains node *N3* with branches *B4.1* and *B4.2*. Apart from sequences and nodes, all branches contain a section labelled “*deltas*”, which records metrics variations in the top level of that branch (i.e. outside any node). If the branch has no sequence (has only nodes) all deltas for the branch are 0. Deltas, in combination with a suitable quality model and metrics composition algorithms, are used for estimating the impact on system quality.

We will now describe our vision for the correction methodology, using the example of a hypothetical tool, that implements it.

At runtime, the software engineer will be able to precisely specify as well as prioritize his targeted quality factors. For example, he might favor flexibility over simplicity, but memory efficiency over flex-

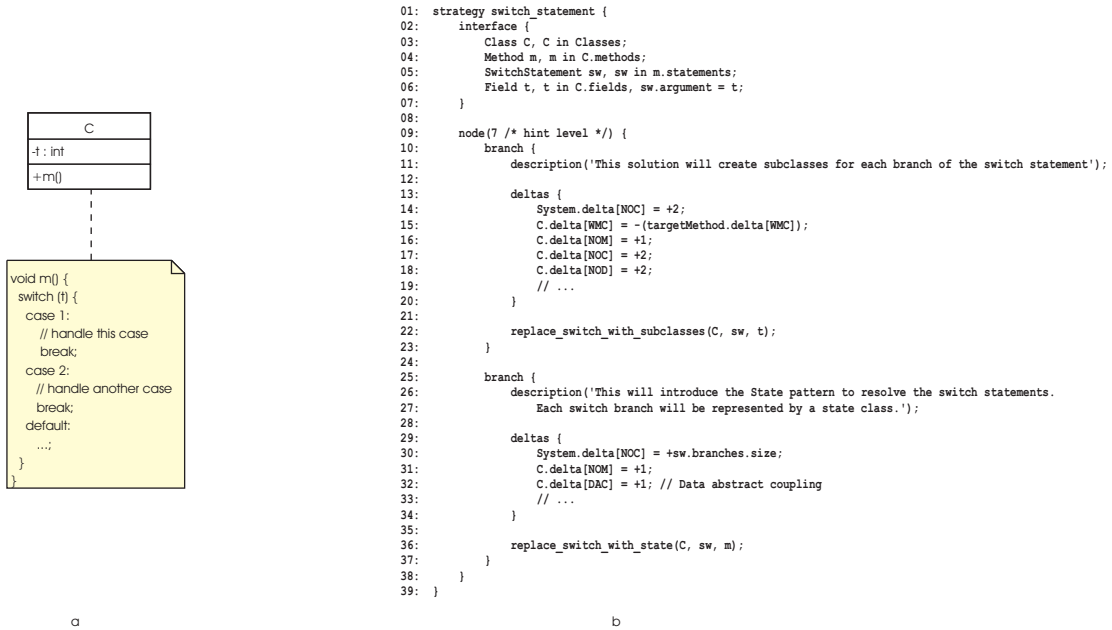


Fig. 1. A design flaw and a possible correction strategy

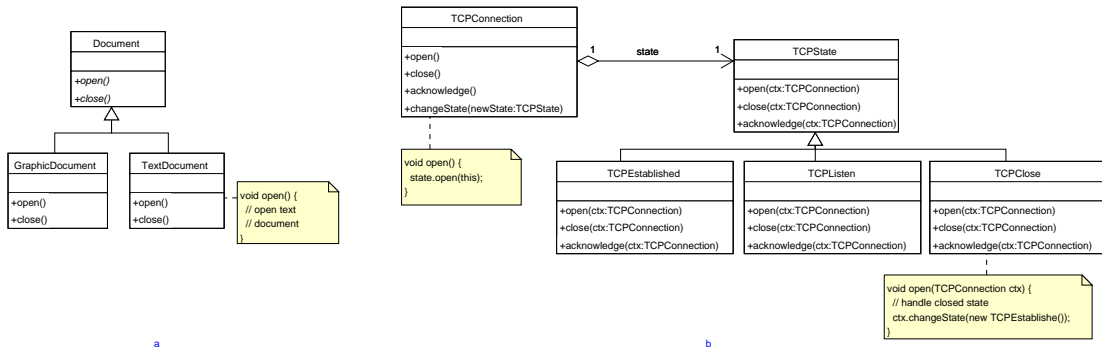


Fig. 2. Two possible ways of avoiding “switch”

ibility. Then, he will define a so called “*hint threshold*”. The “*hint level*” associated to every node in a strategy represents the relative difficulty/risk of the decision that needs to be taken in that node, on a scale from 1 to 10. As the system is being reengineered, the tool compares the threshold to the hint levels associated with encountered nodes. If the hint level is below the threshold, a decision is taken automatically, without asking the user. If the hint level exceeds the threshold, the tool lists all choices together with the estimated quality impact and a recommendation. The estimation and recommendation are computed using the quality model and the individual metric deltas. The quality model is essentially a tree that allows the decomposition of the targeted quality factors into software metrics. Variations in the low level metrics lead to variations in the quality factors.

When faced with such a decision, the engineer has two options: choosing a solution or leaving the decision to the tool, which then will decide automatically. In automatic mode, quality-related information and historic data (previous flaws, previous decisions in the strategy) are used to estimate and decide which of the possible branches is the best one to follow next.

By setting the hint threshold from 1 to 10, the tool traverses a wide range of operation modes, from full-manual (but still with higher automation than today’s refactoring browsers) to full-automatic.

If a certain path that is chosen, either in automatic or manual mode, runs into a dead-end because of unsatisfied preconditions for example, the tool offers two choices: it can either roll back to the previous node in the decision tree and take a different route, or ask the user to eliminate the obstacle and continue on the same route. In figure 3, the thick line

```

strategy ::= interface (sequence|node)+
node     ::= 'node' '{(' hint_level ')}' '{(' branch branch* ')}'
branch   ::= 'branch' '{(' description deltas (sequence|node)+ ')}'
sequence ::= 'sequence' '{(' preconditions transformation+ ')}'
. . .

```

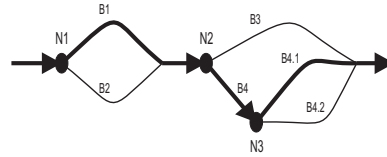


Fig. 3. Illustration of concepts used to define a correction strategy

```

sequence replace_switch_with_subclasses(Class C, SwitchStatement sw, Field t) {
    preconditions {
        foreach (a in assignments(t)) {
            a.owner == C.constructor;
        }
    }

    C.methods.find('switch_method') == nil;
    foreach (branch in sw.branches) {
        System.classes.find(branch.label + C.name) == nil;
        system.interfaces.find(branch.label + C.name) == nil;
    }
}

ClassList branchClasses;
Method targetMethod = sw.owner;

if (m.statements.size > 1)
    targetMethod = extract_method(sw, 'switch_method', 'protected');

foreach (branch in sw.branches) {
    Class branchClass = create_class(branch.label + C.name, C);
    Method newMethod = create_member_function(branchClass, targetMethod);
    add_function_body(newMethod, branch.statements);
    branchClasses.add(branchClass, branch.label);
}
. . .
}

```

Fig. 4. A sample sequence

represents a successful traversal of the strategy and the elimination of the detected design flaw.

It is not entirely clear yet how the estimation algorithms will look like and the degree to which we can talk of local and global optimums with regard to our target quality factors. These issues are currently subject to active research (see section 5).

As shown, the approach will operate on design entities such as classes, methods, interfaces, assignments, etc (*C1*), and is designed to be quality-driven (*C2*). The approach relies on previously diagnosed design flaws (*C3*), without imposing any restriction on their level of abstraction (*C4*). Behavior preservation is ensured with the help of pre and post condition logic (*C5*). Extension of the approach is done through the specification of new correction strategies (*C6*). In this respect, we intend to specify a basic catalog of strategies to common design problems along with a methodology for extending it. It is furthermore obvious that automation is a central point to our approach (*C7*). Finally, correction strategies inherently record reengineering expertise, in the form of corrective measures that can be applied to eliminate a certain design flaw (*C8*).

4 Related Work

In [1] a number of best practices in reengineering large object-oriented systems is identified and formulated in the form of reengineering patterns. More

recently, they have been also published in [5]. The idea behind these patterns is to provide a means of recording and transmitting reengineering expertise (*C8*). Although important and useful, they have the disadvantage that they are informal and do not model the impact on quality in any way. Their purpose is to disseminate best practices, not be a support for future tools. A similar effort, although not in the specific context of large object-oriented software reengineering, is represented by some of the anti-patterns described in [4].

The approaches presented in [10], [11] and [12] are similar in the sense that all of them try to insert design patterns to places in the source code where such patterns are missing, or present in a distorted form. They aim to improve quality this way, although without any formal model to support it. They generally rely on the assumption that design patterns are good (from a qualitative standpoint), and therefore everything else must be “bad”. Criteria *C2*, *C3* and *C4* are not fulfilled. Moreover, the approach presented in [12] starts from the premise that a “set of historically related java programs” (several versions) is available, a premise which is in most of the times false. Finally, the paper employs the term “strategy”, a most unfortunate choice considering that it denotes nothing more than a linear sequence of refactorings. In our opinion this cannot be called a *strategy*.

In [13], refactoring opportunities are detected in source code, using bad smells. The approach however fails to consider quality in any way. The engineer is ultimately left with the hardest choice of all: which one of the many proposed refactorings should be applied, in what order, and what will be the impact of these refactorings on the quality of the system (*C1*, *C2*, *C7*).

Miceli and others propose in [14] and [15] an approach which tries to estimate the impact that certain transformation produce on software quality by selecting a set of relevant metrics. While interesting from the quality model standpoint, the approach does not fulfill *C3* and *C4*, because it is basically not eliminating design flaws (it doesn't even try to detect them), but rather operates as an optimization technique, simply choosing from a set of predefined transformations in such a way that the increase of quality is maximized.

Also of interest regarding quality issues is the work presented in [16], where an approach for legacy code migration to object orientation is presented. Although the paper does not deal with object-oriented design flaw detection and correction, the approach towards modelling the impact of some very low level source transformations on quality is very promising.

Significant work concerning tool support for automated introduction of design patterns has been done in [17] and [18]. Concerning low-level tool support for generic code transformations, we refer the reader to [19].

5 Conclusion and Open Issues

In the beginning of the paper we highlighted the main deficiencies of today's state of the art in the field of object oriented restructuring, most notably the gap that exists between detection and correction technologies. We then stated a set of criteria that define a proper approach to bridge this gap. Next, the idea for a quality-driven, highly automatable approach has been proposed, with special emphasis on our new *correction strategies* and related concepts. As suggested by their name, "strategies" are not programs that list the exact steps to be taken, but plans that guide a (semi)automated tool through a sequence of decisions and their corresponding implementation in the subject system. To a certain degree, correction strategies can be regarded as formalizable reengineering patterns which model the impact on the targeted quality factors of the restructuring process, thus allowing optimal or close to optimal decisions to be taken and executed automatically. It is important to note that we do not try to maximize overall system quality, but rather a restricted set of

quality factors (the targeted quality factors of restructuring), that simplify the remaining phases of the reengineering process.

It is still an open issue whether we can talk about global or local quality optimums concerning restructuring, and the relation that may exist between the two. In other words, we need to give an answer to the question: How can we prove (or can we prove?) that a sequence of (quasi)local optimal decisions leads us to globally optimal or close-to-optimal targeted quality factors. What is the relation between global system quality and the individual design flaws that are detected? Furthermore, how should we coordinate correction strategies? How do design flaws influence each other and what (if any) influence does the order in which detected flaws are eliminated have on the targeted quality factors? Further open issues are how to express our quality factors in terms of low level software metrics, and how to treat cases in which parts of the system are frozen, or impossible to change because of some external constraints.

Our immediate future work will focus on developing a quality model and estimation algorithms, as well as on refining the formalism used to represent correction strategies. A catalog of such correction strategies is also a high priority.

In order to validate the approach in practice, we intend to develop a prototype tool and perform some case studies on real systems. It will be based on, and glue together existing tools, such as: Recoder [20] and Inject/J [19], as infrastructure for source code analysis and transformation, Goose [7] and Prodeos [8] for the detection of design flaws.

References

1. Bär, H., Bauer, M., Ciupke, O., Demeyer, S., Ducasse, S., Lanza, M., Marinescu, R., Nebbe, R., Nierstrasz, O., Przybilski, M., Richner, T., Rieger, M., Riva, C., Sassen, A.M., Schulz, B., Steyaert, P., Tichelaar, S., Weisbrod, J.: The FAMOOS object-oriented reengineering handbook (1999)
2. Fowler, M.: Refactoring. Improving the Design of Existing Code. Addison-Wesley (1999)
3. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: A taxonomy. *IEEE Software* **7** (1990) 13–17
4. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons (1998)
5. Demeyer, S., Ducasse, S., Nierstrasz, O.: *Object Oriented Reengineering Patterns*. Morgan Kaufmann (2003)
6. Riel, A.J.: *Object-Oriented Design Heuristics*. First edn. Addison-Wesley (1996)
7. Ciupke, O.: Automatic detection of design problems in object-oriented reengineering. In: *Proceedings of*

- Technology of Object-Oriented Languages and Systems - TOOLS 30. (1999) 18–32
8. Marinescu, R.: Measurement and Quality in Object-Oriented Design. PhD thesis, "Politehnica" University of Timișoara (2002)
 9. Naur, P.: Revised report on the algorithmic language algol 60. Communications of the ACM **3** (1960) 299–314
 10. Guéhéneuc, Y.G., Albin-Amiot, H.: Using design patterns and constraints to automate the detection and correction of inter-class design defects. In: Proceedings of the TOOLS 39. (2001) 296–305
 11. Jahnke, J., Zündorf, A.: Rewriting poor design patterns by good design patterns. In: ESEC/FSE '97 Workshop on Object-Oriented Reengineering. (1997)
 12. Jeon, S.U., Lee, J.S., Bae, D.H.: An automated refactoring approach to design pattern-based program transformations in java programs. In: Proceedings of the Ninth Asia-Pacific Software Engineering Conference, IEEE (2002)
 13. Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proceedings of 7th European Conference on Software Maintenance and Reengineering, Benevento, Italy. (2003)
 14. Miceli, T., Sahraoui, H.A., Godin, R.: A metric based technique for design flaws detection and correction. In: Proceedings of the International Conference on Automated Software Engineering, IEEE (1999) 307–310
 15. Sahraoui, H., Godin, R., Miceli, T.: Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In: Proceedings of the International Conference on Software Maintenance 2000, IEEE (2000) 154–162
 16. Zou, Y., Kontogiannis, K.: Quality driven transformation compositions for object oriented migration. In: Proceedings of the Ninth Asia-Pacific Software Engineering Conference, IEEE (2002)
 17. Schulz, B., Genssler, T., Mohr, B., Zimmer, W.: On the computer-aided introduction of design patterns into object-oriented systems. In: Proceedings of the 27th TOOLS conference. (1998)
 18. Cinnéide, M.Ó., Nixon, P.: A methodology for the automated introduction of design patterns. In: Proceedings of the IEEE International Conference on Software Maintenance, IEEE (1999) 463–472
 19. The Inject/J team: The Inject/J website. (<http://injectj.sourceforge.net>)
 20. The Recoder team: The recoder website. (<http://recoder.sourceforge.net/index.html>)
 21. Coad, P., Yourdon, E.: Object-Oriented Design. Second edn. Prentice Hall, London (1991)