Pattern-based Refactoring of Legacy Software Systems

Sascha Hunold[1], Björn Krellner[2], Thomas Rauber[1], Thomas Reichel[2], and Gudula Rünger[2]

[1] University of Bayreuth, Germany,
{hunold,rauber}@uni-bayreuth.de,
[2] Chemnitz University of Technology, Germany,
{bjk,thomr,ruenger}@cs.tu-chemnitz.de,

Abstract. Rearchitecturing large software systems becomes more and more complex after years of development and a growing size of the code base. Nonetheless, a constant adaptation of software in production is needed to cope with new requirements. Thus, refactoring legacy code requires tool support to help developers performing this demanding task. Since the code base of legacy software systems is far beyond the size that developers can handle manually we present an approach to perform refactoring tasks automatically. In the pattern-based transformation the abstract syntax tree of a legacy software system is scanned for a particular software pattern. If the pattern is found it is automatically substituted by a target pattern. In particular, we focus on software refactorings to move methods or groups of methods and dependent member variables. The main objective of this refactoring is to reduce the number of dependencies within a software architecture which leads to a less coupled architecture. We demonstrate the effectiveness of our approach in a case study.

Key words: Pattern-based Transformation, Legacy System Restructuring, Business Software, Class Decoupling, Object-oriented Metrics

1 Introduction

The problem of maintaining legacy software is more relevant than ever since many companies are facing the problem of adapting their product lines to new technologies and to short release cycles. During the evolution of software systems new requirements are brought up and old specifications change. In many cases the original software has been built by developers who have left the company years ago. In another scenario, the software architecture has to be reorganized since design decisions have to be adapted by the current developers who very often do not have a complete overview of the entire software. Eick et al. denote this process as *code decay* [6]. The necessary restructuring or rearchitecturing of software systems is a cost-intensive and error-prone task with a high risk of failure when not planned in detail. Some popular software development techniques like

agile software development or *extreme programming* try to reduce these risks by integrating refactoring and restructuring in the development process [2].

In addition to the actual development process, tool support is required to perform software rearchitecturing tasks especially to minimize risk of errors by using automated transformations. Modern integrated development environments (IDEs) support the developer with various transformations (mainly refactorings) or source code generation from user-defined templates (e.g., user interface builders, code completion). Most tools do not enforce a clean separation of concerns when developing an architecture for a specific design pattern, for instance. These tasks have to be realized by the developer on his own.

Rauber and Rünger proposed an incremental transformation process which addresses the problem of restructuring a monolithic business software [14]. This process consists of three steps which have to be traversed to transform the monolithic legacy software into a distributed and modular software system. In the first step (extraction phase), the source code of the legacy system is parsed and transformed into a language independent representation which is called Flexible Software Representation (FSR). The FSR captures all relevant parts of the code structure (e.g., classes or functions) and their dependencies. Furthermore, the source code is annotated to uniquely identify constructs of the programming language in the model. In the second step (transformation), the software is transformed into a high level abstraction layer, preferably using a model driven approach for refactoring. In the last step (generation), the target code is generated using the annotated legacy source code and iteratively applying the defined transformation operations from step 2.

Applying the transformation process mentioned above to real legacy software systems raises several new questions. A very common problem is that the original developer of some code is not known. Thus, it is extremely time-consuming for other developers to figure out what a piece of code is meant to be doing. For this reason, we propose a pattern-based transformation process to improve the software quality automatically. Since quality of source code is hard to measure we rely on software metrics such as code coupling metrics. We consider a software system as improved when software entities are less coupled. Code coupling metrics are based on the number of dependencies between software entities. Therefore, removing dependencies between entities can improve the legacy software since loosely coupled code is easier to modify or to adapt to new requirements.

The proposed automatic transformation process consists of two steps. In the first step, the code of the legacy software system is scanned for predefined patterns to identify the bad smells of software design. This pattern is represented as a graph. The abstract syntax tree (AST) of a legacy program contains the corresponding call graph. The ASTs are scanned in order to find a pattern match. If a pattern is found in the graph, a predefined transformation rule is applied to perform the actual syntax change. It is required that the semantic rules should stay exactly the same. Since this is hard to prove most transformation rules are relatively simple but hard to find by hand.

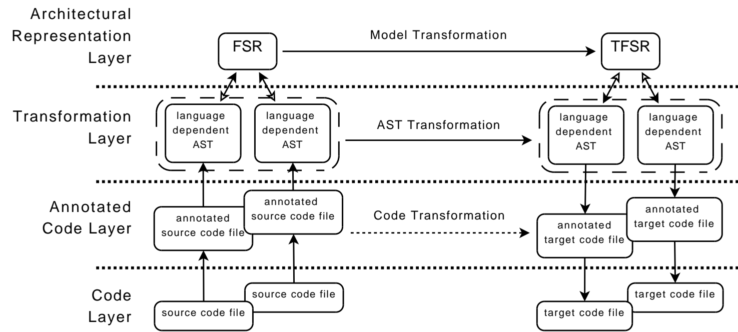

Fig. 1. Abstraction layers of TransFormr. Upwards: model extraction; sidewards: model and code transformation; downwards: code generation.

The contribution of this article is a novel pattern-based transformation process for legacy software systems that can help to automatically remove dependencies between entities. This in turn leads to an improved software architecture. The rest of the paper is organized as follows: Section 2 outlines the incremental transformation process and describes the toolkit TransFormr which implements this process chain. Furthermore, we describe the information which has to be captured within the intermediate representation in order to perform a pattern-based software transformation. Section 3 introduces our automatic approach for software refactoring using pattern-based modifications of ASTs. Its effectiveness is shown by applying the pattern-based refactoring to an example project. Section 5 discusses related work and Section 6 concludes the article.

2 Legacy software transformation

2.1 Incremental software transformation

Software systems can be classified into different software categories like numerical libraries, operating systems, or business software. Each of these classes requires special strategies and methods in order to perform a software rearchitecturing, e.g., migrating to new operating systems or integrating new technologies. We consider the case of monolithic business software systems. Such a business software consists of a single application with several graphical or text-based user interfaces and a database system. A major goal of our work is to create and implement a software transformation process which helps to decompose a legacy software into modules. A modular description of a software makes it possible to perform all kinds of different transformations, e.g., to port the system to a distributed platform, to integrate new features, or to substitute several modules by more efficient implementations.

The software transformation process is divided into three steps [9]. The first step is the *extraction phase* in which the legacy code is converted into an abstract

software model. The structure of the source code is analyzed, e.g., the relationship between classes and methods. Moreover, semantic information (comments, package information) is extracted if possible and attached to the software model. The abstract software model is captured using the flexible software representation. The FSR is an intermediate language to express the structure and the relationships of the source code in a language-independent way. In order to uniquely identify elements of code, e.g., variables or methods, the source code gets annotated in the extraction phase. A major challenge of this step is the categorization of the legacy code into predefined categories such as UI-related code, business logic code, or database-related code. These categories are helpful for later transformations and perception of poorly located functionality. This abstract software model (FSR) enables the software transformation on a higher abstraction level.

The second step is the *transformation phase*. Starting with the FSR, multiple transformations of miscellaneous categories can be applied to the software system. Transformations can vary from simple refactorings (like renaming) to complex ones, like integrating web services. The transformations can be divided into the following categories:

- **basic transformations:** refactorings like rename, move, and create,
- **filter transformations:** to select certain functionality to be kept in the final product,
- **composite transformations:** to relocate functionality onto remote servers, see [9] for more details.

Applying the transformations incrementally leads to the so-called Target FSR (TFSR).

In the last step, the *generation phase*, source code for the target platform is generated from the TFSR. In this step, it is important to generate and to reuse as much code as possible to reduce risks of introducing new bugs.

2.2 TransFormr

The toolkit TransFormr [9] supports the incremental transformation process. The toolkit guides the developer through all phases of the transformation process (extraction, transformation, generation). Figure 1 depicts the abstraction layers which are traversed during the transformation process using the TransFormr toolkit. The source code of the legacy business system is located in the code layer. The annotated code layer contains the code base enriched with TransFormr annotations and forms the basis of the higher abstraction layers. The transition from annotated to executable source code is done by removing all annotation information. The transformation layer is comprised of abstract syntax trees of the annotated source code. These ASTs are then traversed and stored into the flexible software representation which is basically a language-independent description of the language-dependent ASTs. The FSR is located at the top of the abstraction model, combines all information about the software system, and holds references to the source code in order to perform transformation operations.

All transitions between the described layers are performed with a language transformation processor (LTP). We chose TXL [3] as LTP to annotate and to extract the model from the legacy code. It is also used to generate the target code from the model. TransFormr has also been extended to parse comments associated with classes, methods, variables, or statements. This semantic information can be used during the model extraction to separate syntactic and semantic information in the software model.

Most of the transformation operations have to be applied manually by the software architect, i.e., the developer has to select which refactoring operations should be executed. In order to support the architect, TransFormr provides several views on the software, e.g., showing dependencies of a class subset. The visualizations of the software structure, e.g., class, call, or statement dependency diagrams, as well as several software metrics can help observing and evaluating the incremental changes and consequences during the transformation process. We use Coupling Intensity (CINT) and Coupling Dispersion (CDISP) metrics [10] as well as metrics of the following collections: Metrics for Object-Oriented Design (MOOD), Metrics for Object-Oriented Software Engineering (MOOSE), and Quality Metrics for Object-Oriented Design (QMOOD), summarized in [13]. All are variably adapted to our intermediate software model.

All FSR elements contain links to their extracted semantic information (e.g., comments, categorization), for the visualization and transformation. In the generation stage, the information is exported as comments according to comment guidelines, i.e., it is inserted at the appropriate places in the generated source code.

3 Pattern-based moving of MemberGroups

The support for detecting and moving some functionality within legacy code remains a key issue for decoupling software modules. The TransFormr toolkit addresses this problem by providing a pattern-based search to detect separated concerns in classes and several ways to move method code and member variables between classes. Since moving static or global methods and variables around can be easily done, the present work mainly focuses on relocating member methods or member variables.

In the following, two major types of move operations of member methods are considered:

- **Delegation** copies the header and body of the method into another class and replaces the old method's body by a call to the new target method. All references to the old method stay unmodified.
- **Explicit Moving** means that in addition to the moving of program code all references to the method are replaced by an appropriate call to the new method in the target class.

Both move operations have in common that the signature of the moved method has to be altered if public methods of the source class are accessed

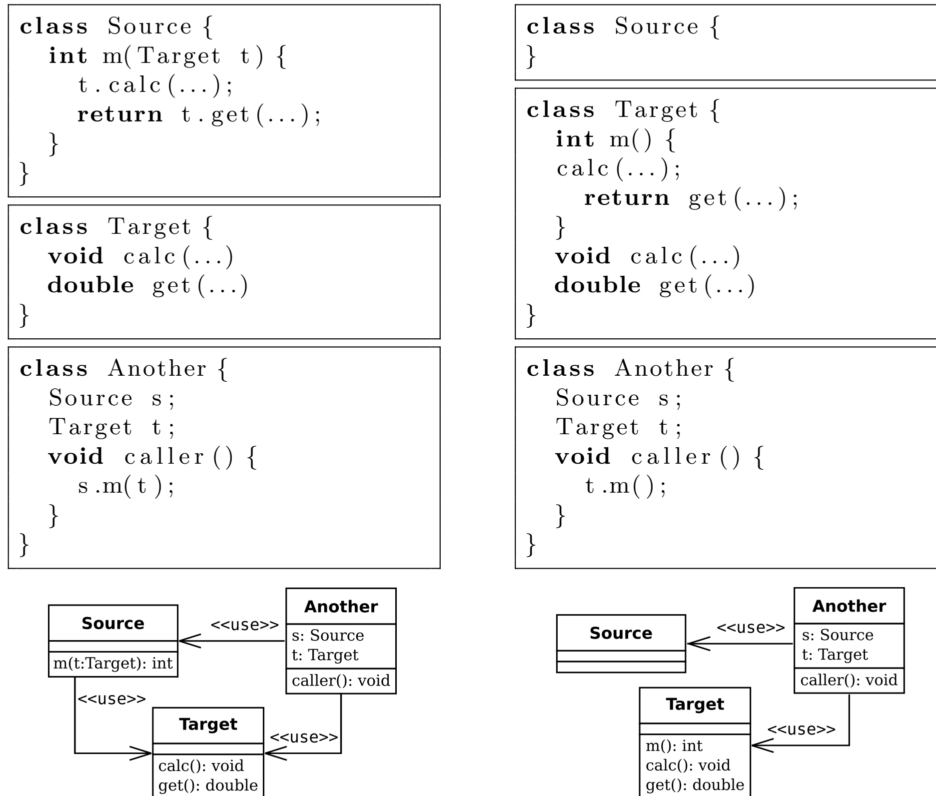

Fig. 2. Example of moving the method `m()` into the parameter class `Target`. Left: initial class model, right: class model after moving `m()`.

within the method. In that case, a new parameter is added to the method which passes a reference to the source class. The move operations cannot be performed if the method to be moved accesses private members of the source class because they are not accessible from the target class. In that case, the visibility of the accessed members has to be changed to overcome this problem.

Delegation is often used if the old class tends to be too complex even if the method is semantically located in the proper class. To reduce complexity and inner class coupling the functionality is moved into a newly created class which should not be visible to other classes in the system as they still call the original method.

If a method is moved explicitly, the functionality of the method should be inserted into a class which fits best. The main limitation of this operation is that a reference to the target class is needed wherever the moved method was called on the source class previously.

A variation of the explicit moving is to make the method a member function of one of its parameters. This special moving can be applied if the coupling of

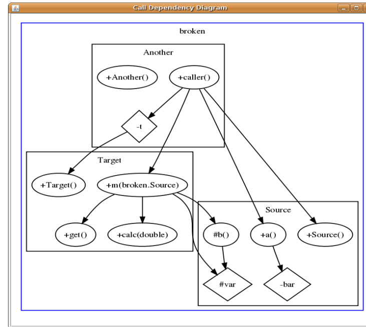

Fig. 3. Broken encapsulation of class `Source` (outward edges of `Target` to former private members of `Source`).

the method to the parameter class is bigger than to the own class. The following example (Figure 2) demonstrates this case. The method `Source:m(Target)` is tightly coupled with class `Target` because it calls only methods of `Target`. Moving `m(Target)` into class `Target` is obvious in this case. Additionally, the parameter `t` is eliminated and the reference `s.m(t)` is replaced with `t.m()` in class `Another`. This procedure is applicable in all cases and is automatically done by IDEs, like Eclipse or NetBeans, for trivial cases like the example above. If the method which should be moved contains dependencies to other members, like the use of a private member variable, the refactoring engines of the IDEs fail or break encapsulation by rising the visibility of private variables (see Figure 3).

To address these problems, we propose a refactoring strategy which helps to detect groups of methods and member variables which have the same concerns and to move these groups between classes. We introduce the term *MemberGroup* to denote such a group. A *MemberGroup* consists of exactly one public method that can access other private members (methods or member variables) of the same class which are not used by other methods. *MemberGroups* often occur if a method's task is split into sub-tasks that are implemented by a couple of private methods and use private member variables of the parent class. If we want to move the public method of the *MemberGroup*, it is useful to move all members (the public method and all accessed private members) of the *MemberGroup* to capture the whole concern. In this paper, we primarily consider members which are not inherited from superclasses, overridden by child classes, or implement methods of an interface. The preconditions for moving those members are not affected by the architectural constraints of the class design.

Based on the description, we propose a strategy that supports the pattern-based transformation process.

1. Build a software model (FSR) of the legacy system with the toolkit TransFormr.
2. Search for *MemberGroups* inside of the classes with graph patterns on the FSR of the software.

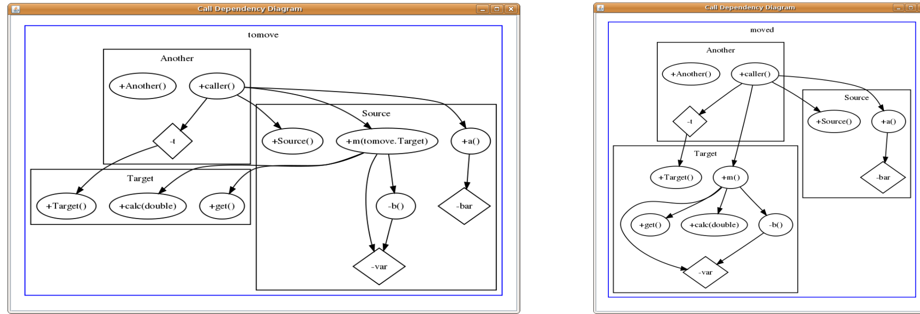

Fig. 4. Left: Example of strong class coupling which can be reduced by moving *MemberGroup* (`m(Target), b(), var`). Right: Class dependencies after moving the *MemberGroup*.

3. For each *MemberGroup*: Present possibilities to move the *MemberGroup* and indicators for each one.
   a) Move into parameter class: Class coupling of *MemberGroup* is bigger to parameter class than to original class.
   b) Delegate *MemberGroup*: Could be used if the public method implements or extends an existing one.
   c) Explicit move: The class coupling to another class is bigger than to the original class.
4. Validate preconditions and apply the transformations on the software model.

To move a *MemberGroup* with delegation (3b) or explicitly (3c), manual interaction is necessary to obtain the reference to the target class in each class in which the *MemberGroup* is used. In case of (3a) the strategy can be performed fully automated and used to reduce the Class Coupling in order to improve the understandability and maintainability of a legacy system (see Section 4).

Code metrics are used to indicate the usefulness of moving a *MemberGroup* to some target class. The metrics are based on the number of outward and inward edges in the call dependency graph. The Coupling Intensity (CINT) [10] metric is defined as the number of distinct method calls from a given method (outward edges). The Coupling Dispersion (CDISP) is defined as the number of classes in which a method is called (number of inward edges) divided by CINT.

Based on the ideas of [10] we introduce indicators to find target classes for *MemberGroups*.

- Move into a parameter class C_p if the *MemberGroup* has more than one edge to C_p. If more than one class is available, use the class with the highest number of edges from the *MemberGroup* to C_p.
- Move the *MemberGroup* into the class with the highest CDISP value and:
  - Use delegation if the *MemberGroup* is semantically correct placed but in a too complex or oversized class or if the public method of the *MemberGroup* implements or extends an existing one;
  - Otherwise use explicit moving.

Table 1. Code metrics for the *MemberGroup* (`m(Target), b(), var`).

Class	CINT	CDISP	Inward Edges
`Target`	2	0	0
`Another`	0	-	1

Automatic moving of a *MemberGroup* is not always possible, e.g., if the public method of the *MemberGroup* implements an interface method. In such a scenario, the developer can be supported by presenting call dependency diagrams with depicted *MemberGroups* and indicators for target classes in order to perform the code change manually.

4 Experimental analysis

In this section, we propose an example of moving a *MemberGroup* into a parameter class applying the strategy described in the previous section. Figure 4 depicts an example of strongly coupled classes on the left hand side. The coupling can be removed by moving the *MemberGroup* (`m(Target), b(), var`) to the parameter class `Target`. An indicator for moving the *MemberGroup*, is the number of outward edges from the *MemberGroup* to other classes (two edges to class `Target` vs. no edges to `Source` and `Another`).

When we apply the strategy described in Section 3 to the example in Figure 4, the graph search finds the *MemberGroup* pattern (`m(Target), b(), var`) in class `Source`. In order to find possible target classes into which the *MemberGroup* can be moved several metrics are calculated, summarized in Table 1. Based on the calculated metrics and the number of inward and outward edges of the *MemberGroup*, we suggest `Target` as target class because of CINT(`Target`)=2 and since `Target` is a parameter of method `m(Target)`.

The result of moving the *MemberGroup* into `Target` is shown in Figure 4 (right). The major improvement after moving the *MemberGroup* is the decoupling of the classes `Source` and `Target`. This coupling improvement can be measured with the class coupling metric (CC) of the MOOD metrics set [1]. Class coupling metric is defined as the ratio of the sum of the class pair couplings $c(C_i,\ C_j)$ and the overall number of class pairs in a system of n classes. The coupling $c(C_i, C_j) = 1$ if there is a dependency between C_i and C_j (method call or variable), otherwise 0.

$$\text{CC} = \frac{\sum_{i=1}^{n} \sum_{j=1, i \neq j}^{n} c(C_i, C_j)}{n^2 - n}$$

The use of the metric in the example results in the following improvement in the class coupling metric:

For legacy applications a lower coupling is desired since a higher coupling increases complexity, reduces encapsulation and potential reuse, and limits understandability and maintainability [1].

	CC
before moving *MemberGroup*	$\frac{3}{6} = 0.5$
after moving *MemberGroup*	$\frac{2}{6} = \mathbf{0.33}$

In a separate study we decomposed the source code of the Apache Jakarta project JMeter[1] (809 classes, 70 kLOC) into *MemberGroups*. A total of 206 of these *MemberGroups* with exactly one public method and at least one private member were found. Due to the fact that the detected *MemberGroups* have to match certain constraints the proposed class transformation could not be applied. Sample constraints of such a transformation are: (a) all classes which hold a reference to the source class also have to hold a reference to the target class, or (b) all methods within a *MemberGroup* must not be part of an interface. Even though no target class could be found for this particular case, the study shows that TransFormr can be used to decompose a software project into *MemberGroups* and that it checks the necessary constraints to apply particular transformations.

5 Related work

The use of patterns is a fundamental principle of software engineering. In contrast to our work, in which we try to exploit design patterns of the legacy software, it is also suitable to use patterns when building a software architecture from scratch. A pattern-based approach for the development of a software architecture is presented in [5]. The main idea of this work is to break down the software design problem into several subproblems and to apply a software pattern (called pattern frames) to solve the subproblems. This makes it possible to change certain design decisions during the evolution of the software by, e.g., instantiating a different design pattern for the implementation of a subproblem.

Fowler et al. introduced many refactorings and design patterns for object-oriented languages as solutions for common mistakes in code style in order to make the software easier to understand and cheaper to modify [8]. The proposed manual changes in the software design are supported by tests to verify the correctness of the software. Other work describes the need for automatic support during refactoring and restructuring tasks but also state the limits and drawbacks of full automated restructuring (e.g., untrustworthy comments, meaningless identifier names) [12].

As in our approach, metrics can be used to suggest possible target classes to move functionality [7]. In contrast to our work, the authors move only single methods and present an Eclipse plug-in to detect code that suggests refactorings (bad smells) in Java projects. Based on a distance metric between classes and methods the plug-in suggests methods to move if the distance to another class is lower than to the original class. The authors applied their approach to two

[1] http://jakarta.apache.org/jmeter/

projects mentioned in [8], and conclude that the plug-in was able to detect a great amount of bad smells which Fowler et al. suggested for these projects as well. Thus, distance or coupling metrics can help to detect misplaced methods [11].

A reengineering methodology for a given software is proposed in [4]. It consists of three steps: create a source code representation (Program Representation Graph (PRG)), transform this representation, and generate target code. The approach defines orthogonal code categories with a concern (user interface (UI), business logic, or data), roles (definition, action, and validation), and controls as connectors between concerns. The categorization process is mainly driven by the categorization of a set of base classes into the concerns (e.g., GUI library classes) followed by a categorization of variables, attributes, and procedures which use the already categorized set. Based on the categorized PRG the authors outline a general move method transformation to detect methods with different concerns in order to separate UI code from data access code.

6 Conclusions

In this article, we have presented a novel approach to perform automated transformations of legacy software. The main goal of the proposed procedure is to support developers by changing the software architecture of a legacy system. We focus on obtaining a better separation of concerns by removing class dependencies automatically. The transformation procedure works as follows: The developer defines or selects a legacy software pattern. This pattern represents a dependency graph. The legacy software is searched for occurrences of this pattern. If the legacy pattern is found a pre-defined target pattern is inserted. As example pattern the *MemberGroup* move pattern was introduced. A *MemberGroup* consists of the publicly accessible class method and its dependent private member methods and class members. An algorithm is presented which finds *MemberGroups* in a legacy system and suggests appropriate target classes based on code coupling metrics. An experimental evaluation shows by example how legacy code can be improved if the proposed transformation method is applied. To justify this pattern-based refactoring process, it is also shown that the legacy patterns considered here can actually be detected in real world software systems.

In future work, we plan to extend the *MemberGroup* move pattern to capture more functional concerns in legacy software. One possible enhancement could be to weaken the restrictions on the visibility of the group members, e.g., dependent methods could also have a public modifier. Another possible pattern could be to identify multiple dependent public methods, e.g., getter and setter functions.

Acknowledgment

The transformation approach described in this article as well as the associated toolkit are part of the results of the joint research project called TransBS funded by the German Federal Ministry of Education and Research.

References

1. F. Abreu and R. Brito. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In *Proc. of the 4th Int. Conf. on Software Quality (ASQC)*, McLean, VA, USA, 1994.
2. Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
3. James R. Cordy. Source transformation, analysis and generation in TXL. In *Proc. of the 2006 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM'06)*, pages 1–11, New York, NY, USA, 2006.
4. R. Correia, C. Matos, M. El-Ramly, R. Heckel, G. Koutsoukos, and L. Andrade. Software Reengineering at the Architectural Level: Transformation of Legacy Systems. Technical report, University of Leicester, 2006.
5. Isabelle Côté, Maritta Heisel, and Ina Wentzlaff. Pattern-based Exploration of Design Alternatives for the Evolution of Software Architectures. *Int. Journal of Cooperative Information Systems, World Scientific Publishing Company*, Special Issue of the Best Papers of the ECSA '07, December 2007.
6. Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.
7. M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. JDeodorant: Identification and Removal of Feature Envy Bad Smells. In *Proc. of the 23rd IEEE Int. Conf. on Software Maintenance (ICSM 2007)*, pages 519–520, Paris, France, October 2007.
8. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Melrose, Massachusetts, 1999.
9. S. Hunold, M. Korch, B. Krellner, T. Rauber, T. Reichel, and G. Rünger. Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing. In *Proc. of the 32nd IEEE Int. Computer Software and Applications Conf. (COMPSAC 2008)*, pages 303–310, Turku, Finland, 2008.
10. Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
11. Mika V. Mäntylä and Casper Lassenius. Drivers for software refactoring decisions. In *Proc. of the 2006 ACM/IEEE Int. Symp. on Empirical Software Engineering (ISESE'06)*, pages 297–306, New York, NY, USA, 2006.
12. Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
13. Lisboa Portugal and Lúcia Baroni. Formal Definition of Object-Oriented Design Metrics. Master's thesis, Ecole des Mines de Nantes, France; Universidade Nova de Lisboa, Portugal, 2002.
14. T. Rauber and G. Rünger. Transformation of Legacy Business Software into Client-Server Architectures. In *Proc. of the 9th Int. Conf. on Enterprise Information Systems*, Funchal, Madeira, Portugal, 2007.