Correction of High-Level Design Defects with Refactorings

Naouel Moha, Saliha Bouden and Yann-Gaël Guéhéneuc

GEODES - Group of Open and Distributed
Systems, Experimental Software Engineering

Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
E-mail: {mohanaou, boudensa, guehene}@iro.umontreal.ca

Abstract

We define design defects as "poor" design solutions that hinder the maintenance of programs. Thus, their detection and correction are important to improve the maintainability and reduce the cost of maintenance. The detection of design defects has been actively investigated by the community. However, their correction still remains a problem to solve. We propose a first method to correct these defects systematically using refactorings. Then, we introduce some challenges that our community must meet.

Keywords: Software Defects, Design Defects, Antipatterns, Detection, Correction, Object-Oriented Architecture.

1 Introduction

Automatic detection and correction of design defects in object-oriented architectures are important to improve software quality, to ease the maintenance of object-oriented architectures, and thus to reduce the cost of maintenance. Indeed, programs free of design defects are easier to change and thus to maintain.

We define design defects as "poor" design choices that hinder the maintenance of programs. They include bad solutions to recurring problems in object-oriented design, such as antipatterns [3] (as opposed to design patterns [6]), defects related to design patterns (abusive or ill-advised uses), and code smells [5] (symptoms of design defects). We consider antipatterns as high-level design problems as opposed to code smells, which are low-level problems.

We proposed a systematic method to specify design defects consistently and precisely [11] and to generate detection algorithms from their specifications automatically. We specify a language based on rules that allows to define these specifications with structural, semantic, and measurable properties that characterize a design defect. This method was a first step towards the systematic detection of design defects.

The detection of design defects is one problem but their correction is yet an issue to be solved. The solution advocated for correcting design defects is to apply refactorings [3 ; 5 ; 13]. Refactoring is a technique used to change *"the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* [5]. However, refactorings proposed to solve defects in the literature can be performed manually and are difficult to automate since they depend on the context and the environment. We propose a method based on a language that allows to specify the refactorings to apply for correcting design defects. Both languages for the detection and the correction of design defects are based on a meta-model that aims to represent programs at different levels of abstraction. Our meta-model specifies entities such as a class, a method, a relationship, with which we can build representations of programs.

Section 2 introduces related work. Section 3 presents the meta-model we designed. The two following Sections 4 and 5 present respectively the detection and the correction of design defects. We finally conclude this paper by stating challenges remaining in correction of design defects.

2 Related Work

Refactoring was first introduced in 1992 by Opdyke in his thesis [12]. Refactorings can be applied at the design level (high level or composite refactorings) or at the code level (low level or primitive refactorings). High level refactorings can be performed by combining

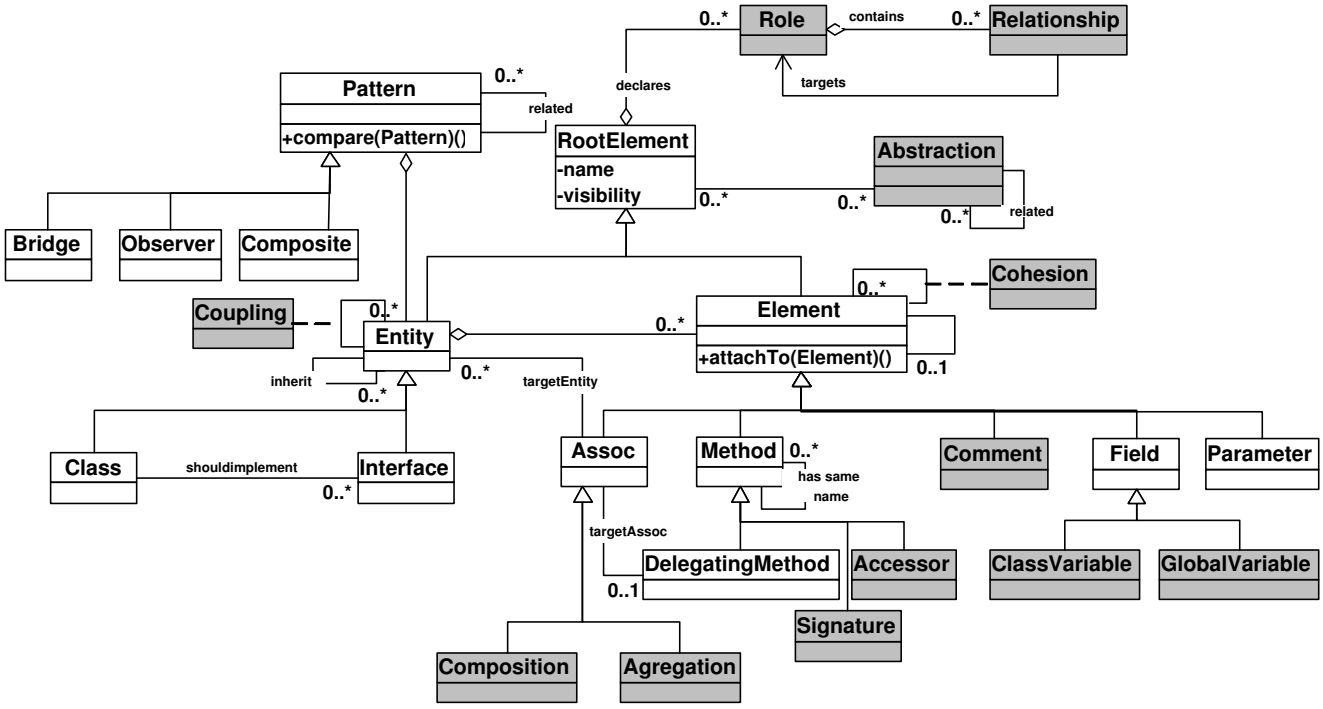

Figure 1. PADL Meta-model.

several low level refactorings. Opdyke defined 26 primitive refactorings and 3 composite refactorings and their associated preconditions, to ensure the behavior preservation after their application. According to Opdyke, as long as each primitive refactoring preserves the behavior, then the result of the transformation of the composition preserves the behavior.

Opdyke has also participated to the book of Martin Fowler [5], which describes more refactorings to perform but in an informal and manual manner. Fowler proposed a catalogue of refactorings. In one of the chapter of his book, Kent Beck contributed by describing how to find bad smells in code and how to clean them up with refactorings. Bad smells are low-level design defects in the source code of a program suggesting that maintainers should apply refactorings [5].

Tichelaar *et al.* defined a language-independent meta-model, named FAMIX, for representing object-oriented programs and for performing refactorings; they also presented a feasibility study concerning primitive refactorings for Smalltalk and Java [14]. FAMIX consists of entities including Class, Method, and Attribute. This meta-model is supported by the Moose Refactoring Engine, which is part of the Moose Reengineering Environment [4], a tool environment for reengineering object-oriented systems.

Several tools such as *Refactoring Browser* [9], *XRefactory* [16], or *Eclipse* are available to perform refactorings. They allow to automate refactorings in an easy and quick manner. They provide an environment to improve the structure of programs, and thus, reduce the cost of reusability of software.

3 A Meta-model for the Detection and the Correction of Design Defects

Our meta-model PADL (*Pattern and Abstract-level Description Language*) [1] is a language-independent meta-model to model object-oriented programs and the structure of the solutions of design patterns [2], including binary class relationships [8] and accessors. PADL offers a set of constituents (classes, interfaces, methods, fields, relationships, and so on) with which we can build models of programs (*cf.* Figure 1). It offers also methods to manipulate these models easily and to generate other models, using the Visitor design pattern. In this work, we choose PADL because it is mature (5 year-old) and is actively maintained in-house.

Our meta-model distinguishes between classes and interfaces, and also member classes. Thus, we do not need additional analysis to determine if we are dealing with Java classes or Java interfaces as with FAMIX [14]. Moreover, contrary to FAMIX, we do not

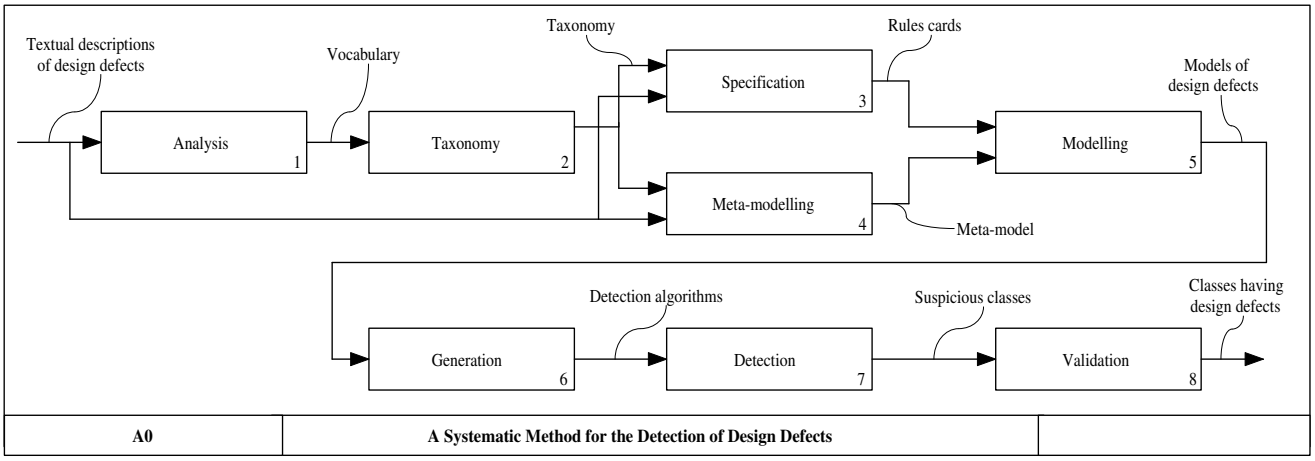

Figure 2. Method for the Description and Detection of Design Defects.

represent constructors as methods. We also differentiate global variables from usual attributes. However, the meta-model FAMIX allows to perform refactorings, which is not yet fully supported by our meta-model. We are currently extending it with methods to perform refactorings for the correction of design defects.

4 Detection of Design Defects

For the detection of design defects, we build a PADL model of the program and apply our detection algorithms on this model. The meta-model PADL provides enough accurate information to get efficient results for the detection of design defects since it enables us to well specify structural, semantical, and measurable properties related to design defects. Figure 2 illustrates our method for the detection of design defects. We proposed a systematic method to obtain precise specifications of *any kind of* design defects and to generate detection algorithms from these specifications. Our method decomposes in 8 steps [10].

1. **Analysis.** We extract key concepts from the textual descriptions of design defects in the literature. Key concepts include metric-based heuristics as well as structural and semantic information. Key concepts form a consistent vocabulary of reusable concepts to describe design defects, with no synonym and homonym.

2. **Taxonomy.** We define a taxonomy of the described design defects by classifying their key concepts in disjoint (sub)categories. This taxonomy is a reference model to distinguish design defects, to highlight their commonalities and specificities, and thus, to avoid misunderstanding and misinterpretations.

3. **Specification.** We specify design defects as sets of rules (and compositions thereof) in rule cards, using the vocabulary and the taxonomy of design defects. Rule cards express literary descriptions of design defects synthetically with a unified vocabulary and in a precise form following a dedicated BNF grammar. A rule card specifies the structural relationships among the roles and characterises roles according to their semantics (names), structural and measurable properties. Roles can be played by any source code constructs, i.e., classes, methods, parameters, and so on. For lack of space, we cannot provide the rule cards of all these design defects. However, they are available with the material for replication at `http://ptidej.iro.umontreal.ca/downloads/experiments/propASE06/`.

4. **Meta-modelling.** We enrich the meta-model PADL to instantiate rule cards of design defects and thus get (in step 4) models of design defects with which to generate automatically detection techniques. Our meta-model for design defects reifies the key concepts used to specify the defects as rule cards.

5. **Modelling.** Using the constituents of the SADDL meta-model, we instantiate the rule cards to build concretely models of design defects that can be manipulated programmatically. The set of modelled design defects forms a catalogue, which is the basis for the automatic detection techniques.

We validate manually the models of design defects with respect to their original literary forms and associated rule cards, thus ensuring that we can describe design defects using the constituents of the SADDL meta-model appropriately. During this step, we may correct and enrich the meta-model to improve the descriptions of design defects. This step is important to refine iteratively the analysis of key concepts, the taxonomy, the specification, the meta-model, and the models of design defects.

At the end of these first 5 steps, we obtain *precise definitions*, in the form of rule cards and programmatic models, of *any kind of* design defects, including code smells and *antipatterns*, in terms of both metrics and *structural* information. In the two remaining steps:

6. **Generation.** We generate algorithms to detect the modelled design defects in programs using the constituents of the meta-model (and associated rule cards and key concepts). The algorithms are based on metric values, on semantic properties, and on the structures of the programs. The detection source code is based on our SAD framework (Software Architectural Defect), which provides all the services required to write detection algorithms of de- sign defects for AOL [2], C++, and Java programs, be it manually or automatically.

7. **Detection.** We build a PADL model of the program and apply the detection algorithms on this model, using the services provided by the SAD framework. The detection algorithms return lists of suspicious classes.

8. **Validation.** We validate the detection algorithms by looking for design defects in open-source programs and by manually analysing the results. The validation is performed manually, because only developers can assess the validity of detected design defects.

We thus obtain detection algorithms systematically from the informal descriptions of design defects in the literature. We apply these algorithms on *open-source* programs, which generates a first library of manually validated design defects for future comparisons and replications.

5 Correction of Design Defects

We propose to extend our previous method for the correction of design defects. We focus on high-level design defects such as antipatterns. An antipattern [3] is a literary form describing a bad solution to a recurring design problem, which has a negative impact on the quality of a program architecture. Code smells, which are low-level defects, are possible symptoms of higher-level defects.

Like the factorization in mathematics of a complex expression in a combination of several simple expressions, our method aims at specifying design defects in a combination of code smells, and similarly, to refactor a complex defect in a sequence of small refactorings of code smells.

We enumerate succinctly each step of this first method for correcting a high-level design defect (*cf.* Figure 3).

1. **Analysis.** A review of the literature in refactoring is necessary to determine which refactorings to correct which design defect.

2. **Taxonomy.** We propose to define a taxonomy of refactorings by distinguishing the primitive and composite refactorings and also the refactorings that are localized in a class than those that involve several classes. This decomposition highlights the extent of the code inspection required to correct a design defect and the spread of the changes in programs caused by the refactorings.

3. **Specification.** We propose to define a language based on rules to correct design defects. We did not yet specify this language but we plan to use the same syntax of the language for specifying rules of detection or an extension of thereof. However, we have a good idea to how to structure. Indeed, high-level design defects, such as antipatterns, are constituted of a set of code smells, we propose to associate to each code smell a set of refactorings. These refactorings can be either primitive or composite. We propose to perform refactorings on code smells first because they are small defects and second because they are less prone of errors. The sequence of the refactorings associated to code smells correct the whole design defect.

4. **Meta-modelling.** We enrich and use the meta-model PADL to instantiate rules for detecting design defects and thus get (in step 4) models of design defects with which to generate automatically correction techniques. We are currently working on this step and the next one.

5. **Modelling.** We instantiate the rules to build concretely the models of the solutions of design defects i.e., design defects refactored.

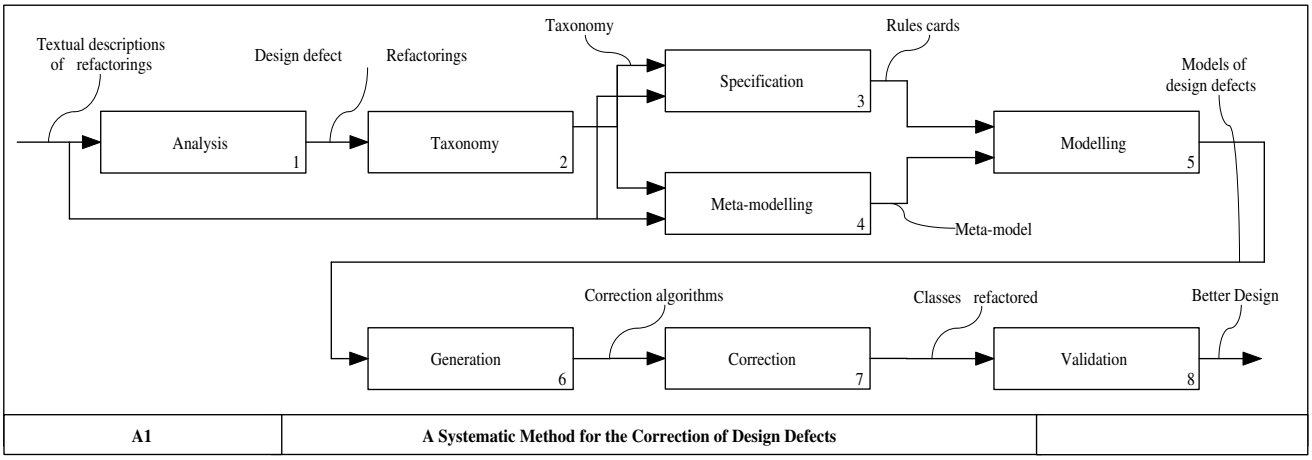

Figure 3. Method for the Description and Correction of Design Defects.

6. **Correction.** This step consists in applying the correction algorithms and propose to the user the solutions for the design defects. We suggest to developers the list of refactorings to apply and the solutions expected to improve their design.

7. **Validation.** The developer can approve or not the refactorings suggested.

This process is iterative since the application of refactorings to correct defects can introduce new defects. The preview offers the possibility to the developer to abort the refactoring since it does not improve the design.

It is important to highlight that the refactorings that we propose to apply are at the model level and not in the code level. Indeed, we apply refactorings on models of programs, which are instantiated from the metamodel PADL. However, we plan to apply these refactorings on the code level and perform tests to ensure that the application of refactorings have not introduce new errors.

5.1 Illustrative Example

Let assume, we want to apply our method to correct the Blob design defect.

The Blob (called also God class [13]) corresponds to a large controller class that depends on data stored in surrounded data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolises most of the processing done by a program, takes most of the decisions, and closely directs the processing of other classes [15]. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

The Blob is composed of mainly two code smells: Large Class and Data Class. According to Fowler, a large class can be refactored by performing the following refactorings: Extract Class [5, p.149] and Extract Subclass [5, p.330].

The refactoring Extract Class specifies that if "*You have one class doing work that should be done by two, create a new class and move the relevant fields and methods from the old class into the new class.* [5, p.149]" So, Fowler suggests to create a new class and move the fields and the methods of the large class in the new class using the two primitive refactorings: Move Field [5, p.146] and Move Method [5, p.142]. However, in the case of the design defect Blob, instead of creating a new class to store extra fields and methods of the large class, the developer would attempt to store these fields and methods in the surrounded data classes. Indeed, why the developer would create new classes, if there are already classes that lack of behavior such as data classes. As proposed by Brown [3], the refactored solution of the Blob consists in identifying cohesive sets of methods and fields that represent the same abstractions. Then, we need to move these sets to surrounded data classes. Thus, it simplifies the large class and adds some behavior to the simple data classes. This refactoring provides a better object-oriented design as the previous suggestion of Fowler.

5.2 A Language for Specifying Refactorings

As described by Fowler, refactorings have to be performed manually and adapted according to the context

```
RULE_CARD: Blob {
    RULE: Blob {ASSOC: associated FROM: LargeClass ONE TO: DataClass MANY};
    RULE: LargeClass {   ExtractClass
                       || ExtractSubclass };
    RULE: ExtractClass  { if (METRIC: LCOM, HIGH) then MoveMethod(LargeClass,DataClass)
                                                    && MoveField(LargeClass,DataClass) };
};
```

Figure 4. Simplified Rule Card of Correction of the Blob.

of the application and the intentions of the developers, and so, these refactorings are difficult to automate. To overcome this limitation, we propose to specify refactorings in primitive rules to automate them and avoid different interpretations. The advantage of a language is that it enables the developers to specify easily rules to perform according to the context and the type of programs. The specification in rules enables to automate refactorings, re-apply and reuse them for other programs. We propose a formalisation of these rules in what we call *rule cards*.

5.3 Formalisation of Rule Cards

A rule card describes a design defect, its code smells and their properties, and the relationships among code smells. It allows also to specify the refactorings to correct code smells. We formalise rule cards with a BNF grammar. A BNF grammar determines the exact syntax for a language. For a lack of space, we did not present this grammar, but the reader can find a description in [10]. Figure 4 illustrates the grammar with the rule card of the Blob design defect.

A rule card is identified by the keyword `RULE_CARD`, followed by the name of the code smell and a set of rules specifying this specific design defect as a set of code smells and the associated refactorings. The Blob design defect is divided in two main code smells: Large Class and Data Class. These two code smells represent classes tied by an association relationship (`ASSOC`). The code smell `LargeClass` can be corrected with one of the following refactorings: Extract Class and Extract Subclass. The third rule specifies that if in a class there is a lack of cohesion (LCOM) high, we move the methods and fields from the large class to the data classes.

6 Conclusion

This position paper presented succinctly the method we specified to detect design defects. We extended this method for the correction of design defects. We illustrated our approach with the Blob defect.

In our team, we are currently working on refactoring techniques in the code and the design level. We organise in a catalog all refactorings that are possible to implement by specifying the pre- and post-conditions and the list of elementary actions associated to each refactoring. Actually, we are implementing these refactorings in our framework PTIDEJ [7].

As the detection, the correction of defects will be based on models of programs. Thus, it is interesting to consider and explore graph and model transformation techniques (the model can be seen as a graph).

Here is a list of challenges that our community must meet :

- Define a language for specifying the rules for correcting design defects.
- Exploit benefits offered by model transformation techniques and refactorings. We will apply them based on the transformation rules defined for each defect.
- Extend our approach to the code level.

References

[1] Hervé Albin-Amiot, Pierre Cointe et Yann-Gaël Guéhéneuc. Un méta-modèle pour coupler application et détection des design patterns. Michel Dao et Marianne Huchard, éditeurs, *actes du* 8^e *colloque Langages et Modèles à Objets*, volume 8, numéro 1-2/2002 de *RSTI – L'objet*, pages 41–58. Hermès Science Publications, janvier 2002.

[2] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: A novel approach and a case study. In Tibor Gyimóthy and Vaclav Rajlich, editors, *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Computer Society Press, September 2005. Best Paper Award.

[3] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, March 1998. ISBN: 0-471-19713-0.

[4] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *CoSET '00: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, June 2000.

[5] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999. ISBN: 0-201-48567-2.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[7] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[8] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[9] The Refactory Inc. Refactoring browser, October 1999.

[10] Naouel Moha and Yann-Gaël Guéhéneuc. A systematic method for the detection of design defects. In *OOPSLA'06 : Object-Oriented Programming, Systems, Languages and Applications*, October 2006. Submitted.

[11] Naouel Moha, Duc-Loc Huynh et Yann-Gaël Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. Roger Rousseau, éditeur, *actes du* 12^{e} *colloque Langages et Modèles à Objets*. Hermès Science Publications, March 2006.

[12] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[13] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[14] Sander Tichelaar, Stephane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of ISPSE '00 (International Conference on Software Evolution)*, pages 157–167. IEEE Computer Society Press, 2000.

[15] Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsabilities and Collaborations*. Addison-Wesley Professional, 2002. ISBN: 0201379430.

[16] XRef. *XRefactory (Previously Named XRef-Speller)*, 2000.