Analysing Anti-patterns Static Relationships with Design Patterns

Fehmi Jaafar*, Yann-Gaël Guéhéneuc*, and Sylvie Hamel**
* PTIDEJ Team, DIRO, Université de Montréal, QC, Canada
** LBIT Team, DIRO, Université de Montréal, QC, Canada
E-Mails: jaafarfe@iro.umontreal.ca, yann-gael.gueheneuc@polymtl.ca, hamelsyl@iro.umontreal.ca

***Abstract*—Anti-patterns are motifs that are commonly used by developers but they are ineffective and counterproductive in program development and–or maintenance. These motifs evolve and they may have dependencies with non-anti-pattern classes. We propose to analyse these dependencies (in particular with design patterns) in order to understand how developers can maintain programs containing anti-patterns. To the best of our knowledge, no substantial investigation of anti-pattern dependencies with design patterns was presented. This paper presents the results of a study that we performed on three different Java systems (ArgoUML, JFreeChart, and XercesJ) to analyse the static relationships between anti-patterns and design patterns. We detect such static relationships to better understand software systems and to explain the co-existence of these motifs. Our finding provides evidence that developers encapsulate anti-patterns using design patterns to facilitate maintenance tasks and reduce comprehension effort.**

***Keywords*—Anti-patterns, Design Patterns, Static Relationships, Mining Software Repositories.**

I. Context and Problem

Software systems continuously evolve in order to incorporate changing customers' requirements, performance improvements and–or to fix bugs. Without proper knowledge, developers may introduce anti-patterns in the system. An anti-pattern is a literary form that describes a bad solution to recurring design problems that leads to negative effects on code quality [1]. Opposite to anti-patterns, design patterns [2] are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to changes. Large, long-lifespan systems often have both design patterns and anti-patterns and, consequently, anti-patterns and design patterns may have some relationships, *i.e.*, the classes participating in some design patterns may be in relation to those participating in some anti-patterns. The static relationships among classes are typically use, association, aggregation, and composition relationships [3].

Research Problem. Most previous work agree that anti-patterns make the maintenance of systems more difficult and that design patterns can serve as guide in program exploration and, thus, ease maintenance. However, there are no investigation in the literature about the static relationships among anti-patterns and design patterns. Yet, understanding these relationships help developers better understand and maintain software systems by giving them the knowledge of what is "good" and what is "bad" in their system and, thus, what and how they could strive to eliminate design defects.

We formulate the following research question:

- Are there static relationships between anti-patterns and design patterns?

We found that, for the majority of anti-patterns, we detect static relationships among design patterns with different proportions in ArgoUML, JFreeChart, and XercesJ. We discuss this finding to explain the existence and the distribution of these relationships among anti-patterns.

Organisation. Section VI relates our study with previous work. Section II presents our method to detect instances of classes participating in patterns and anti-patterns. Section III describes our empirical study. Section IV presents the study results, while Section V discusses them, along with threats to their validity. Finally, Section VII concludes the study and outlines future work.

II. Approach

This section describes the steps necessary to extract and to analyse the data required to perform this study. We use two previous approaches DECOR[4] to detect anti-patterns, and DeMIMA[5] to detect design patterns.

A. Step 1: Detecting Anti-patterns

We use the Defect DEtection for CORrection Approach DECOR[4] to specify and detect anti-patterns. DECOR is based on a thorough domain analysis of anti-patterns defined in the literature, and provides a domain-specific language to specify code smells and anti-patterns and methods to detect their occurrences automatically. It can be applied on any object-oriented system through the use of the PADL [5] meta-model and POM framework. PADL is a meta-model to describe object-oriented systems [5]. POM is a PADL-based framework that implements more than 60 metrics. Moha *et al.* [4] reported that the DECOR current detection algorithms for anti-patterns ensure 100% recall and have a precision greater than 31% in the worst case, with an average precision greater than 60%.

B. Step 2: Detecting Design Patterns

We use the Design Motif Identification Multilayered Approach (DeMIMA)[5] to specify and detect design patterns. DeMIMA ensures traceability between motifs and source code by first identifying idioms related to binary class relationships to obtain an idiomatic model of the source code and then by using this model to identify design motifs and generate a design model of the system. We also use DeMIMA to detect motifs's relationships. In fact, DeMIMA distinguishes use, association, aggregation, and composition relationships because such relationships exist in most notations to model systems, for example, in UML. Gueheneuc and Antoniol [5] reported that the DeMIMA approach detection for patterns ensures 100% recall and have a precision greater than 34%. While, for the detection of relationships among classes, the DeMIMA approach ensures 100% recall and precision.

C. Step 3: Analysing Anti-patterns Dependencies

Table I summarizes anti-patterns considered in this paper. To perform the empirical study, we choose to analyse the relationships of the well known anti-patterns and six design patterns belonging to three categories: creational patterns (Factory method and Prototype), structural patterns (Composite and Decorator), and behavioral patterns (Command and Observer). Thus, we choose these motifs because they are representative of problems with data, complexity, size, and the features provided by classes. We choose, also, these motifs because they have been used and analysed in previous work [4] and [6]. Definitions and specifications are outside of the scope of this paper and are available in [2] and [6].

We assume that a design pattern *P* has a static relationships with the anti-pattern *A* if at least one class belonging to *P* has a use, association, aggregation, or composition relationships with one class belonging to *A*.

III. Study Definition and Design

The *goal* of our study is to investigate the static relationships between anti-patterns and design patterns in the system. The *quality focus* is explaining anti-patterns' static relationships with design patterns to better understand systems' architectures. The *context* of our experiment is three open-source Java programs: ArgoUML, JFreeChart, and XercesJ.

A. System Under Analysis

We apply our approach on three Java systems: ArgoUML[1] (version 0.26), JFreeChart[2] (version 1.0.6), and XercesJ[3] (version1.0.4). These systems can be classified as large, medium, and small systems, respectively. We use these systems because they are open source, have been used in previous work, are of different domains, and have between hundreds and thousands of classes. Table I summarizes some statistics about these systems.

[1]http://argouml.tigris.org/
[2]http://www.jfree.org/
[3]http://xerces.apache.org/xerces-j/

	ArgoUML	JFreeChart	XercesJ
# of classes	3,325	1,615	1,191
# of AntiSingleton	3	38	24
# of Blob	100	49	12
# of CDSP	51	3	6
# of ComplexClass	158	52	7
# of LongMethod	336	75	7
# of LongParameterList	281	76	4
# of MessageChains	162	59	8
# of RefusedParentBequest	123	5	7
# of SpaghettiCode	1	2	6
# of SpeculativeGenerality	22	3	29
# of SwissArmyKnife	13	26	29

Table I
DESCRIPTIVE STATISTICS OF THE OBJECT SYSTEMS (CDSP: CLASSDATASHOULDBEPRIVATE)

IV. Study Results

We now present the results of our empirical study. Table II summarizes our results.

RQ1: Are there static relationships between anti-patterns and design patterns?

Yes. Table II shows that, for the majority of anti-patterns, we detect static relationships with design patterns. On the one hand,we notify that different anti-patterns can have different proportions of static relationships with design patterns. This observation is not surprising because these systems have been developed in three unrelated contexts, under different processes. On the second hand, the design pattern that often has the most relationships with anti patterns is the Command design pattern. For example, we noted that 50% of static relationships among SpeculativeGenerality and design patterns in ArgoUML, are with the Command motifs. In XercesJ, we observe that 41% of relationships among ClassDataShouldBePrivate was with the Command design pattern.

No clear tendency exists for ComplexClass and RefusedParentBequest. For example, ComplexClasses have static relationships with six analysed design patterns with equivalent proportions in ArgoUML, JFreeChart, and XercesJ.

But... In the three systems, if a class participate in a design pattern, it does not have a relationship with the SpaghettiCode anti-pattern, as showed in Table II. In fact, in all three systems, we do not detect any class playing role in a SpaghettiCode having static dependencies (use, association, aggregation, and composition relationships) with the six design patterns (Command, Composite, Decorator, FactoryMethod, Prototype, and Observer).

Relevance. Design patterns are especially geared to improve

Anti-patterns	Systems	# of SR
AntiSingleton	ArgoUml	68
	JFreeChart	92
	XercesJ	83
Blob	ArgoUml	161
	JFreeChart	72
	XercesJ	42
ClassDataShouldBePrivate	ArgoUml	83
	JFreeChart	31
	XercesJ	44
ComplexClass	ArgoUml	182
	JFreeChart	84
	XercesJ	66
LongMethod	ArgoUml	212
	JFreeChart	290
	XercesJ	142
LongParameterList	ArgoUml	290
	JFreeChart	188
	XercesJ	204
MessageChains	ArgoUml	192
	JFreeChart	94
	XercesJ	77
RefusedParentBequest	ArgoUml	146
	JFreeChart	72
	XercesJ	48
SpaghettiCode	ArgoUml	0
	JFreeChart	0
	XercesJ	0
SpeculativeGenerality	ArgoUml	20
	JFreeChart	34
	XercesJ	67
SwissArmyKnife	ArgoUml	35
	JFreeChart	84
	XercesJ	86

Table II
PROPORTION OF THE RELATIONSHIPS OF ANTI-PATTERNS WITH DESIGN PATTERNS (SR: STATIC RELATIONSHIP AMONG ANTI-PATTERNS AND DESIGN PATTERNS)

adaptability and maintainability. Each design pattern aims to make specific changes easier [7]. Thus, the benefit of using design patterns to correct anti-patterns is realised only if we detect and we analyse relationships among them. For example, in XercesJ, the class `org.apache.xerces.validators.common.XMLValidator.java` is an excessively complex class interface. The developer attempts to provide for all possible uses of this class. In her attempt, she adds a large number of interface signatures to meet all possible needs. The developer may not have a clear abstraction or purpose for `org.apache.xerces.validators.common.XMLValidator.java`, which is represented by the lack of focus in its interface. Thus, we claim that this class belongs to a SwissArmyKnife anti-pattern. This anti-pattern is problematic because the complicated interface is difficult for other developers to understand and obscures how the class is intended to be used, even in simple cases. Other consequences of this complexity include the difficulties of debugging, documentation, and maintenance. We detect that this class has a use-relationship with the class `org.apache.xerces.validators.dtd.DTDImporter.java`, which belongs to the Command design pattern. Using Command classes makes it easier to delegate method calls without knowing the owner of the method or the method parameters. Thus, developer can correct `org.apache.xerces.validators.common.XMLValidator.java`, by using the related Command pattern, to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method, and values for the method parameters. Thus, by using the relationships of an anti-pattern with a specific design pattern, we explain how developers maintained the anti-pattern classes while reducing its influence on the system. An external information from the changelog file support this idea (see Section V).

V. DISCUSSION

This section discusses the results reported in Section IV as well as the threats to their validity.

A. Observations

From Table II, we note that many anti-patterns in ArgoUML, JFreechart, and XercesJ have relationships with design patterns. To the best of our knowledge, we are the first to report these relationships, thanks to our use of state-of-the-art detection algorithms, which detects occurrences of 11 anti-patterns and six design patterns. Moreover, we do not consider that an anti-pattern is necessarily the result of a "bad" implementation or design choice; only the concerned developers can make such a judgement. We do not exclude that, in a particular context, an anti-pattern can be the best way to actually implement and–or design a (part of a) class. For example, automatically-generated parsers are often very large and complex classes. Only developers can evaluate their impact according to the context: it can be perfectly sensible to have these large and complex classes if they come from a well-defined grammar.

SpaghettiCodes have no static relationships (use, association, aggregation, and composition) with design patterns. This observation is not surprising because a SpaghettiCode is revealed by classes with no structure, declaring long methods with no parameters, and using global variables for processing. A SpaghettiCode does not take the advantage of object-orientation mechanisms: polymorphism and inheritance. Many object methods have no parameters, and utilize class or global variables for processing. Thus, a SpaghettiCode is difficult to reuse and to maintain, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse. The findings of our analysis indicate that no relation is detected between the different occurrence of SpaghettiCode anti-pattern and design patterns. However, it could be possible that they have

no relations because they constitute DeadCode. Dead code means unnecessary, inoperative code that can be removed. It is a code in the program which can never be executed or a code that is executed but has no effect on the output of a program [8]. Dead code analysis can be performed using live variable analysis, a form of static code analysis and data flow analysis [9]. However, in large programming projects, it is sometimes difficult to recognize and eliminate dead code [10]. Thus, dead code detection can be performed by mining version-control systems (Concurrent Versions System named CVS[4] and Apache Subversion System named SVN[5]), to identify, for example, which classes were never changed after their introduction in the analyzed systems [11]. We noted for example, that in ArgoUML, more than 80% of classes were maintained three times at most. On the other hand, less than 1% of classes were maintained 50 times at least. Based on change analysis, it is neither possible to conclude that SpaghettiCode classes have no relations with design patterns because they constitute DeadCode nor is the opposite true. Indeed, from the results of this case study it is impossible to definitely exclude the possibility that there is in fact no statistically relevant correlation between SpaghettiCode and DeadCode. However, it could be true that the spaghetti code classes have no dependencies because of the lower number of instances in the analysed systems (9 instances).

We observe, also, that the majority of static relationships among anti-patterns and design patterns come from the Command pattern. This design pattern is implemented as a motif in which an object is used to represent and encapsulate all the information needed to call a method at a later time. Thus, developers use this design pattern, possibly unintentionally, when there is a proliferation of similar methods and the user-interface code becomes difficult to maintain. This characteristic can explain, predominately, the static relationships of this design pattern and the anti-patterns:

- ClassDataShouldBePrivate because commands must access the data of other objects to function, and developers may have used public instance variables to allow this access;
- LongMethod and LongParameterList because commands must access the functionalities provides by other classes, which typically can perform lots of processing, in long methods and-or with long parameter lists;
- SpeculativeGenerality because classes in relation to commands may have been engineered with extension in mind, but the command does not use it.

On the one hand, the notion of static dependency can be used to assess the architecture of a software system. In fact, we can assume that systems with more static relationships among design patterns and anti-patterns are more stables, since these relationships can be explained by a developers recours to a recognized and used stable solutions (design patterns) to correct and refactor design defects. For example, by mining software version-control systems, we found that the design pattern Command described in Section IV and containing the class `org.apache.xerces.validators.dtd.DTDImporter.java` was created by the developer *jeffreyr* on 2000-04-04 15:38:39, to *Factoring Validators code* implemented in `org.apache.xerces.validators.common.XMLValidator.java`. In future work, we plan to investigate the use of such static relationships among design motifs to measure the architectural sturdiness. On the other hand, in this paper we study correlations among collocated anti-patterns and design patterns because there might be an interaction effect that could explain the existence of such motifs. In fact, our results showed that the presence of some anti-patterns (LongMethod, LongParameterList, etc.) may increase the chances of the presence of a specific design pattern (Command design pattern). While, the presence of spaghetti code do not has any direct correlation with the presence of design patterns.

B. Threats to Validity

We now discuss in details the threats to the validity of our results, following the guidelines provided in [12].

Internal validity, in our context, they are mainly due to errors introduced in measurements. We are aware that the detection technique used includes some subjective understanding of the definitions of the anti-patterns and design patterns. However, as discussed, we are interested to relate anti-patterns "as they are defined in DECOR" [4] with design patterns "as they are defined in DeMIMA" [5]. For this reason, the precision of the anti-patterns and design patterns detection is a concern that we agree to accept.

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to replicate our study. Moreover, both ArgoUML, JFreeChart, and XercesJ source code repositories are available. Finally, the data sets on which we computed our statistics are available on the Web[6].

Threats to *external validity* concern the possibility to generalise our observations. First, although we performed our study on three different, real systems belonging to different domains and with different sizes and histories, we cannot assert that our results and observations are generalisable to any other systems and the facts that all the analysed systems are in Java and open-source may reduce this generability. Second, we used particular, yet representative, sets of anti-patterns and design patterns. Different anti-patterns and design patterns could have lead to different results, which are part of our future work.

[4]http://cvs.nongnu.org/

[5]http://subversion.apache.org/

[6]http://www.ptidej.net/download/experiments/msr12/

VI. Related Work

Several work studied the detection and the analysis of anti-patterns and design patterns. For lack of space, we only cite some relevant work, the interested readers can find more references in our previous work [4] and [5].

Anti-patterns Definition and Detection. Code smells are related to the inner workings of classes while anti-pattern include the relationships among classes and are more situated on a micro-architectural level. The first book on "anti-patterns" in object-oriented development was written in 1995 by Webster [13]. In this book, the author reported that an anti-pattern describes a frequently used solution to a problem that generates ineffective or decidedly negative consequences. Riel [14] defined 61 heuristics characterising good object-oriented programming to assess a program quality manually and improve its design and implementation. These heuristics are similar and–or precursor to code smells. Brown *et al.* [1] described 40 anti-patterns, which are often described in terms of lower-level code smells. These books provide in-depth views on heuristics, code smells, and anti-patterns aimed at a wide academic audience. They are the basis of all the approaches to detect anti-patterns.

The study presented in this paper relies on anti-patterns detection approach proposed in [4]. However several other approaches have been proposed in the past. For example, Van Emden *et al.* [15] developed the JCosmo tool. This tool parses source code into an abstract model (similar to the Famix meta-model). It used primitive and rules to detect the presence of smells and anti-patterns. The JCosmo tool can visualize the code layout and anti-patterns locations. The goal of this tool is to help developers assess code quality and perform refactorings. The main difference compared with other detection tools is that JCosmo tries to visualize problems by visualizing the design. Marinescu *et al.* developed a set of detection strategies to detect anti-patterns based on metrics [16]. They later refined their methodologies by collecting information from documentation with problematic structures. They showed how to detect several anti-patterns, such as God Classes and Data Classes. Settas *et al.* explored the ways in which an anti-pattern ontology can be enhanced using Bayesian network [17]. Their approach allowed software developers to quantify the existence of an anti-pattern using Bayesian network, based on probabilistic knowledge contained in an anti-pattern ontology regarding relationships of anti-patterns through their causes, symptoms and consequences.

The Integrated Platform for Software Modeling and Analysis (iPlasma) described in [18] can be used for anti-patterns detection. This platform calculates metrics from C++ or Java source code and applies rules to detect anti-patterns. The rules combine the metrics and are used to find code fragments that exceed some thresholds.

We share with all the above authors the idea that anti-patterns detection is a powerful mechanism to asses code quality, in particular indicating whether the existence of anti-patterns and the growth of their relationships makes the source code more difficult to maintain.

Design Pattern Definition and Detection. The first book on "design patterns" in object-oriented development was written in 1996 by Gamma et al. [2]. Since this book, several workshops and conferences have emerged to propose new patterns. Many papers have been published studying the use, impact of patterns. The study presented in this paper relies on design patterns detection approach proposed in [5]. However several other approaches have been proposed in the past. For example, one of the first papers about detecting design patterns was written by Kramer *et al.* [19] in 1996. They introduced an approach detecting design information directly from C++ header files. This information is stored in a repository. The design patterns are expressed as PROLOG rules which are used to query the repository with the extracted information. Their work focused on detecting five structural design patterns: Adapter, Bridge, Composite, Decorator, and Proxy. Recently, an approach based on similarity scoring has also been proposed [20], which provides an efficient means to compute the similarity between the graph of a design motif and the graph of a system to identify classes potentially design motif. Iacob [21] presented a method aims at identifying proven solutions to recurring design problems through design workshops and systems analysis. Indeed, during a design workshop, a team of 3-5 designers is asked to design a system and the design issues they address are collected. Moreover, a set of systems are analysed in order to identify in what measure the design issues discussed during the workshops are considered in the implementation of existing solutions. Candidates for being documented as design patterns are the most recurring design issues in both the workshops and the systems analysis.

Anti-patterns' Static Relationships analysis. There are few papers analyzing the relationships among anti-patterns and design patterns. Vokac [22] analyzed the corrective maintenance of a large commercial program, comparing the defect rates of classes participating in design motifs against those that did not. He found that the Observer and Singleton motifs are correlated with larger classes; classes playing roles in Factory Method were more compact, less coupled, and less defect prone than others classes; and, no clear tendency exists for Template Method. Their approach showed correlation between some design patterns and smells like LargeClass but do not report an exhaustive investigation of possible correlations between these patterns and anti-patterns. Pietrzak and Walter [23] defined and analysed the different relationships that exist among smells and provide tips how they could be exploited to alleviate detection of anti-patterns. These relations presented concentrate on direct dependencies between smells. They performed an

experiment to show that the use of the knowledge about identified smells in Jakarta Tomcat code supports the detection process. They founded examples of several smell dependencies, including simple, aggregate and transitive support and rejection relation. The certainty factor for those relations in that code suggests the existence of correlation among the dependent smells and applicability of this approach to anti-patterns detection. Rather than focusing on the relationships among code smells and anti-patterns, our study focuses on analysing anti-patterns relationships with design patterns.

VII. Conclusions and Future Work

In this paper, we provide empirical evidence of the relationships between anti-patterns and design patterns. We showed that some anti-patterns are significantly more likely to have relationships with design patterns than other. This study raises a question, within the limits of the threats to its validity, about the conjecture in the literature that some anti-patterns have a negative impact on system architecture. We provide a basis for future research to understand precisely the causes and the eventual consequences of the relationships between anti-patterns and design patterns, *i.e.* if developers use design patterns to encapsulate anti-patterns. The advantages of knowing these relations are (1) spotting how developers strive to maintain a system containing anti-patterns by using design patterns and (2) detecting correlations among collocated anti-patterns and design patterns to identify the causes of the co-existence of such motifs.

Future work includes (i) replicating our study on other systems to assess the generality of our results and (ii) analysing the evolution of the relationships between anti-patterns and design patterns by studying different versions of these systems through times.

Acknowledgements. This work has been partly funded by a FQRNT Team grant, the Canada Research Chair in Software Patterns and Patterns of Software, and the Tunisian Ministry of Higher Education and Scientific Research.

References

[1] W. Brown, H. McCormick, T. Mowbray, and R. Malveau, *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998, vol. 20.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

[3] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the UML cake," in *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, D. C. Schmidt, Ed. ACM Press, October 2004, pp. 301–314.

[4] Naouel Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering (TSE)*, vol. 36, no. 1, January–February 2010, 16 pages.

[5] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008.

[6] F. Khomh, M. D. Penta, Y. gal Guhneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on software changeability," *Empirical Software Engineering*, 2011.

[7] D. Jain and H. J. Yang, "Oo design patterns, design structure, and program changes: An industrial case study," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, pp. 580–.

[8] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM, 1994, pp. 147–158.

[9] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, "A c++ data model supporting reachability analysis and dead code detection," *IEEE Trans. Softw. Eng.*, pp. 682–694, 1998.

[10] F. Damiani and F. Prost, "Detecting and removing dead-code using rank 2 intersection," in *Selected papers from the International Workshop on Types for Proofs and Programs*. London, UK, UK: Springer-Verlag, 1998.

[11] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society.

[12] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*. London: SAGE Publications, 2002.

[13] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995. [Online]. Available: www.amazon.com/exec/obidos/ASIN/1558513973

[14] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[15] E. V. Emden and L. Moonen, "Java quality assurance by detecting code smells," in *in Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer*. Society Press, 2002, pp. 97–107.

[16] D. Ratiu *et al.*, "Using history information to improve design flaws detection," 2004.

[17] D. Settas, A. Cerone, and S. Fenz, "Enhancing ontology-based antipattern detection using bayesian networks," *Expert Systems with Applications*, 2012.

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[19] C. Krmer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Proceeding of the 3rd working conference on reverse engineering*. IEEE Computer Society Press, 1996, pp. 208–215.

[20] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *Transactions on Software Engineering*, vol. 32, no. 11, November 2006.

[21] C. Iacob, "A design pattern mining method for interaction design," in *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS '11. ACM, 2011, pp. 217–222.

[22] M. Vokac, "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Trans. Softw. Eng.*, vol. 30, December 2004.

[23] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," *Extreme Programming and Agile Processes in Software Engineering*, pp. 75–84, 2006.