Discussion on the Results of the Detection of Design Defects

Naouel Moha
Yann-Gaël Guéhéneuc

GEODES - Group of Open and Distributed Systems, Experimental Software Engineering
Department of Informatics and Operations Research
University of Montreal, Quebec, Canada

Laurence Duchien
Anne-Françoise Le Meur

Adam Team – INRIA
Laboratoire d'informatique fondamentale de Lille
Université de Lille, France

E-mail: {mohanaou,guehene}@iro.umontreal.ca and {duchien,lemeur}@lifl.fr

Abstract

Software engineers often need to identify in their systems "poor" design choices—design defects—that hinder the development and maintenance, as opportunities of improvements and as a measure of the quality of their systems. However, the detection of design defects is difficult because of the lack of specifications and tools. We propose DECOR, a method to specify design defects systematically and to generate automatically detection algorithms. With this method, software engineers analyse and specify design defects at a high-level of abstraction using a unified vocabulary and a dedicated language for generating detection algorithms. To illustrate our method, in this paper, we specify 4 well-known design defects, the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife and their 15 underlying code smells and we generate automatically their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall and discuss the precision of these algorithms on 11 open-source object-oriented systems.

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

1 Introduction

Quality is an important goal in the software development process because software are everywhere from game applications to life support systems. Quality is assessed and improved mainly during formal technical reviews, which primary objective is to detect errors or defects early, before they are passed on to another software engineering activity or released to the customer [11]. Quality is especially important during maintenance, which is one of the most difficult and expensive activities of the software development process [11] because bad design practices and architectural drift [10] are the root causes of *design defects*, which make adding, debugging, and evolving features difficult.

We define design defects as bad solutions to recurring problems in object-oriented designs, typically UML class diagrams, similarly to design patterns [5]. They encompass problems at different levels of granularity, ranging from architectural and design problems, such as antipatterns [1] to low-level or local problems, such as code smells [3], which are usually symptoms of more global design defects. Design defects are at a higher-level than Halstead or Fenton's "defects", which are "deviations from specifications or expectations which might lead to failures in operation" [2][7]. We focus on the design defects described in Fowler's and Brown's books [1 ; 3].

An example of design defects is the Spaghetti Code antipattern[1], which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, such as polymorphism and inheritance.

The detection of design defects and their correction early in the development process substantially reduce the cost of subsequent activities of the development and support phases [11] because designs free of defects

[1]This defect, as the ones presented later on, is really in between design and implementation.

are easier to implement, change, and maintain. However, detection in large designs is a highly time- and resource-consuming and error-prone activity [13] because design defects crosscut classes and methods and their descriptions are subject to misinterpretation.

We propose the DECOR method (*Defect dEtection for CORrection*) to specify and to detect design defects systematically[2]. This method builds on and generalises previous work and our own experience in a unified method and understanding. It is a major improvement of our previous work [8 ; 12]. It consists of 7 steps, from the analysis of the textual descriptions of design defects through their reification to their detection.

The originality of the method stems from the choice of defining a domain-specific language: Central to the method is the first language to specify design defects at a high-level of abstraction using the key concepts found in their textual descriptions, not in the underlying ad-hoc modelling or detection frameworks as in previous work. Moreover, this language leads to the definition of the first meta-model to describe design defects and to generate automatically detection algorithms. The language is built from an in-depth analysis of the domain related to design defects to identify key concepts, such as metrics, structural relationships, and lexical and structural properties.

The language is then used to generate automatically detection algorithms for the specified design defects. A thorough validation of the generated detection algorithms shows the precision and recall [4] of the algorithms, which supports the consistency and the precision of the specifications and the usefulness of the DECOR method for software engineers during technical reviews and maintenance. It is the first time such an extensive validation is performed to assess detection algorithms for design defects.

This paper aims only to present the validation of our method with the specification and detection of 4 design defects: Blob, Functional Decomposition, and Swiss Army Knife, on 11 object-oriented systems: ArgoUML, Azureus, GanttProject, Log4J, Lucene, Nutch, PMD, QuickUML, and two versions of Xerces, plus Eclipse. It discusses also the results, concludes, and presents future work.

2 Results

We validate our method by studying both the application of the 7 steps and the results of the detection. We use *reverse-engineered* designs in our validation because industrial designs are seldom available freely. Also, design documents, as documentation in general, are often out-of-date. Thus, in many systems with poor documentation, the source code is the only reliable source of information [9].

First, we compute the precision and recall of the results of our method using data obtained independently. This is the first available report of the precision and recall of the algorithms to detect design defects. Thus, we recast our work in the domain of information retrieval and used the measures of precision and recall, where precision assesses the number of true defects identified, while recall assesses the number of true defects missed by the algorithms with the following definitions [4]:

$$\text{precision} = \frac{|\{\text{existing defects}\} \cap \{\text{detected defects}\}|}{|\{\text{detected defects}\}|}$$

$$\text{recall} = \frac{|\{\text{existing defects}\} \cap \{\text{detected defects}\}|}{|\{\text{existing defects}\}|}$$

We enlisted the help of independent software engineers to compute the recall of the generated algorithms. The validation is performed manually because only software engineers can assess whether a suspicious class is *indeed* a defect or a false positive, depending on the rule card and the context and characteristics of the system. This step may be time consuming if the specifications of the design defects are not constraining enough.

Second, we specify and detect design defects in several object-oriented systems and report the numbers of suspicious classes and the precisions and computation times of the algorithms.

2.1 Assumptions of the Validation

This experimental data allows to test 3 assumptions supporting the usefulness of our method.

1. *The language allows to describe several design defects.* This hypothesis supports the applicability of our method on 4 design defects, composed of 15 code smells, and the consistency of the specifications.

2. *The generated detection algorithms have a recall of 100%, i.e., all known design defects are detected, and an average precision greater than 33%, i.e., the detection algorithms report less than* 2/3 *of false positives with respect to the number of true positives.* This hypothesis supports the precision of the rule cards and the adequacy of the algorithm generation. It also supports the services provided by our framework.

[2]Correction is out of the scope of this paper, thus we focus on the part of the DECOR method related to detection.

3. *The complexity of the generated algorithms is reasonable, i.e., have computation times under few minutes.* This hypothesis supports the precision of the generated algorithms and the adequacy of the SAD framework to describe and to analyse program designs.

2.2 Subjects of the Validation

We use our method to describe 4 well-known but different antipatterns from Brown's book [1]: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. Table **??** summarises each of these design defects. These design defects include in their specifications 15 different code smells described, partly described in Fowler's book [3]. We generate associated detection algorithms automatically.

2.3 Objects of the Validation

The objects of our validation are the reverse-engineered designs of 10 open-source Java systems: ArgoUML, Azureus, GanttProject, Log4J, Lucene, Nutch, PMD, QuickUML, and two versions of Xerces. In contrast with previous work, we use *freely available* systems to ease comparisons and replications of our validation. We provide some information on these systems in Table 1. We also apply the algorithms on Eclipse and discuss the results.

Name	Version	Lines of Code	Number of Classes	Number of Interfaces
ArgoUML	0.19.8	113,017	1,230	67
An extensive UML modelling tool				
Azureus	2.3.0.6	191,963	1,449	546
A peer-to-peer client implementing the BitTorrent protocol				
GanttProject	1.10.2	21,267	188	41
A project-management tool to plan projects with Gantt charts				
Log4J	1.2.1	10,224	189	14
A logging Java package				
Lucene	1.4	10,614	154	14
A full-featured text-search Java engine				
Nutch	0.7.1	19,123	207	40
An open-source web search engine, based on Lucene				
PMD	1.8	41,554	423	23
A Java source code analyser for identifying low-level problems				
QuickUML	2001	9,210	142	13
A simple UML class and sequence diagrams modelling tool				
Xerces	1.0.1	27,903	189	107
A framework for building XML parsers in Java				
Xerces	2.7.0	71,217	513	162
Release of March 2006 of the Xerces XML parser				

Table 1. List of Systems.

2.4 Results of the Validation

We report the times of detections and the number of suspicious classes. Manual code inspections were performed by three master students and independent software engineers to compute the precision and recall of the suspicious classes.

Table 2 presents the precision and recall of the detection of the four design defects in Xerces v2.7.0. We ask several independent software engineers to analyse manually this system using only Brown and Fowler's books and their own understanding to identify design defects. Each time a doubt on a candidate class arose, they considered the books as references in deciding by consensus whether or not this class was actually a design defect. They performed a thorough study of Xerces and produced a XML file containing suspicious classes for the 4 design defects. Few design defects may have been missed by mistake due to the nature of the task. We ask other software engineers to perform this same task again on Xerces to confirm the findings and on other systems to increase our database in future work.

The recalls of our detection algorithms are 100% for each design defect. Precisions are between 41.07% to more than 80%, providing between 5.65% and 14.81% of the total number of classes, which is reasonable for a software engineer to analyse by hand, with respect to analysing the entire system—513 classes—manually.

Table 3 provides the numbers of suspicious classes, first line of each row, the numbers of true defects, second lines, the precisions, third lines, and the computation times, fourth lines, for the 9 other systems plus Xerces v2.7.0. We only report precisions, recalls are part of the future work because of the required time-consuming manual analyses by independent software engineers.

We perform all computations on a Intel Dual Core at 1.67GHz with 1Gb of RAM. Computation times do not include building the models of the program designs but include accesses to compute metrics and to check structural relationships and lexical and structural properties.

We verify each of the 3 assumptions using the results of the validation to assess the usefulness of our method.

1. *The language allows to describe several design defects.* We described 4 different design defects of the inter- and intra-class categories and of the structural, lexical, and measurable categories which are characterised by code smells also belonging to different categories. Thus, we show that we can describe different kinds of defects, which supports the generality of our method.

2. *The generated detection algorithms have a recall of 100% and an average precision greater than 33%.* Table 2 shows that the precision and recall for

Design Defects	Number of Classes	Numbers of True Positives		Numbers of Detected Defects		Precision	Recall	Time
Blob	513	39	(7.60%)	44	(8.58%)	88.64%	100.00%	2.45s
Functional Decomposition		15	(2.92%)	29	(5.65%)	51.72%	100.00%	0.91s
Spaghetti Code		46	(8.97%)	76	(14.81%)	60.53%	100.00%	0.23s
Swiss Army Knife		23	(4.48%)	56	(10.91%)	41.07%	100.00%	0.08s

Table 2. Precision and Recall in XERCES v2.7.0. (In parenthesis, the percentage of classes affected by a design defect.)

	ArgoUML	Azureus	GanttProject	Log4J	Lucene	Nutch	PMD	QuickUML	Xerces v1.0.1	Xerces v2.7.0	Average Precision
Blob	29 (2.36%) 25 (2.03%) 86.21% 3.01s	41 (2.83%) 38 (2.62%) 92.68% 6.41s	10 (5.32%) 9 (4.79%) 90.00% 2.44	3 (1.59%) 3 (1.59%) 100% 1.34s	3 (1.95%) 2 (1.29%) 66.67% 1.84s	6 (2.90%) 4 (1.93%) 66.67% 3.56s	4 (0.95%) 4 (0.95%) 100% 3.87s	0 (0%) 0 (0%) 100% 0.45s	10 (5.29%) 10 (5.29%) 100% 2.75s	44 (8.58%) 39 (7.60%) 88.64% 2.45s	89.09%
F.D.	37 (3.01%) 22 (1.79%) 59.46% 0.42s	44 (3.04%) 17 (1.17%) 38.64% 0.47s	15 (7.98%) 4 (2.12%) 26.67% 0.80s	11 (5.82%) 6 (3.17%) 54.55% 0.05s	1 (0.65%) 0 (0.00%) 0% 0.03s	15 (7.25%) 3 (1.45%) 20.00% 0.05s	13 (3.07%) 4 (0.95%) 30.77% 0.06s	10 (7.04%) 3 (2.11%) 30.00% 0.02s	4 (2.12%) 4 (2.12%) 100.00% 0.03s	29 (5.65%) 15 (2.92%) 51.72% 0.16s	41.18%
S.C.	44 (3.58%) 38 (3.09%) 86.36% 0.26	153 (15.56%) 125 (8.62%) 81.70% 2.86s	14 (7.45%) 10 (5.32%) 71.43% 0.20	3 (1.59%) 2 (1.06%) 66.67% 0.08s	8 (5.19%) 6 (3.89%) 75.00% 0.09s	26 (12.56%) 22 (10.63%) 84.61% 0.11s	9 (2.13%) 5 (1.18%) 55.56% 0.06s	5 (3.52%) 0 (0.00%) 0% 0.03s	25 (13.23%) 23 (12.17%) 92.00% 0.11s	76 (14.81%) 46 (8.97%) 60.53% 0.22s	67.39%
S.A.K.	108 (8.78%) 18 (1.46%) 16.67% 0.28s	145 (10.00%) 33 (2.27%) 22.76% 0.13s	8 (4.25%) 3 (1.59%) 37.50% 0.05s	51 (26.98%) 33 (17.46%) 64.70% 0.02s	9 (5.84%) 1 (0.65%) 11.11% 0.02s	33 (15.94%) 13 (6.28%) 39.39% 0.02s	13 (3.07%) 6 (1.42%) 46.15% 0.02s	6 (4.22%) 1 (0.70%) 16.67% 0.02s	12 (6.35%) 5 (2.65%) 41.67% 0.03s	56 (10.91%) 23 (4.48%) 41.07% 0.05s	33.77%

Table 3. Results of Applying the Detection Algorithms. (In each row, the first line is the number of suspicious classes, the second line is the number of classes *being* design defects, the third line is the precision, and the fourth line shows the computation time. Numbers in parenthesis are the percentages of the classes being reported. (F.D. = Functional Decomposition, S.C. = Spaghetti Code, and S.A.K. = Swiss Army Knife))

XERCES v2.7.0 fulfill our hypothesis with precisions greater than 40% and a recall of 100%. Table 3 present the precisions for the other 9 systems, which also comply with our hypothesis, with average precisions above 33%, thus validating the usefulness of our method.

3. *The complexity of the generated algorithms is reasonable, i.e., have computation times under few minutes.* Computations times are in general below few seconds because the complexity of our detection algorithms depends only on the number of classes in the analysed system, n, and on the number of properties to verify on each class. The complexity of the generated detection algorithms is $(c + op) \times \mathcal{O}(n)$, where c is the number of properties and op the number of operators.

2.5 Discussions of the Results

The computation times of the design defects vary with the design defects and the systems. During validation, we notice that building the models of the program designs accounted for the major part of the computation times, while the detection algorithms have small execution times, which explains the only slight differences between each system, in a same line in Table 3, and the differences between each design defect, in different columns. The computation times for PADL models is not surprising, because the models contain extensive data on a system, including binary class relationships [6] and accessors.

The number of detected suspicious classes vary with the design defect and the system because the systems have been developed in different contexts and may have unequal quality. Systems such as PMD or QUICKUML may be of better quality than AZUREUS or XERCES, thus leading to greater numbers of suspicious classes being actually defects.

The precisions also vary in function of the design defect and of the system, as shown in Table 3. This variation has two reasons. First, the specifications of the design defects as rule cards can be over- or under-constraining. For example, the rule cards of the Blob and Spaghetti Code design defects specify the defects restrictively using metrics and structural relationships, leading to a low number of suspicious classes and high precisions. On the contrary, the rule cards of the Functional Decomposition and Swiss Army Knife design defects specify these defects loosely, using lexical data, leading to lower precisions. We could now refine the specifications of the defects and improve their precisions by iterating over the different steps of the method systematically.

The reader could argue that the number of false positives is quite high. However, in these experiments, we obtain false positives because our objective was 100% recall for all programs. Yet, thanks to the systematic method and the language, the rules can be refined and modified systematically to fit the specific contexts of the analyzed systems and, thus, increase precision if desired (possibly at the expense of recall because precision and recall are a trade-off).

We also apply our detection algorithms on the ECLIPSE open source development platform to demonstrate its scalability. ECLIPSE v3.1.2 weighs 2,538,774 lines of code for 9,099 classes and 1,850 interfaces. It is one order of magnitude larger than the largest of the open-source systems, Azureus. The detection of the 4 design defects in ECLIPSE requires more time and produce more results. We detect 848, 608, 436, and 520 suspicious classes for the Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife design defects, respectively. The detections take about 1h20m for each defect, with about 1 hour to build the model. The use of the detection algorithms on ECLIPSE shows the scalability of our algorithms. It also highlights the problem of balance between numbers of suspicious classes and precisions. In addition, it shows the importance of specifying design defects in the context of the system in which they are detected. Indeed, the large number of suspicious classes for the Blob design defect in ECLIPSE, about $1/10^{th}$ of the overall number of classes, may come from design and implementation choices and constraints *within* the ECLIPSE community and thus does not necessarily reflect design *defects*. With our method, software engineers can re-specify design defects easily to fit their environment.

2.6 Illustrations of the Results

We present examples of the four design defects succinctly. In XERCES v2.7.0, method `matchCharArray (Context,Op,int,int,int)` of the `org.apache.xerces.impl.xpath.regex.RegularExpression` class is a typical example of Spaghetti Code. A good example of the Blob design defect is the `com.aelitis.azureus.core.dht.control.impl.DHTControlImpl` class in AZUREUS. This class declares 54 fields and 80 methods for 2,965 lines of code. An interesting example of the Functional Decomposition design defect is class `org.argouml.uml.cognitive.critics.Init` of ARGOUML, in particular because the name of the class suggests a functional decomposition. Class `org.apache.xerces.impl.dtd.DTDGrammar` is a striking example of Swiss Army Knife in XERCES, implementing 4 different sets of services with 71 fields and 93 methods for 1,146 lines of code.

2.7 Threats to Validity

A threat to the validity of the validation is the exclusive use of open-source Java systems. It is possible that the open-source development process biases the numbers of design defects, especially in case of mature systems such as PMD v1.8 or XERCES v2.7.0. It is also possible that the Java programming language affects design choices and thus design defects. However, we applied our algorithms on systems of various size and quality to void the possibility for all systems to be either well or badly implemented. Moreover, we chose to perform a validation on open source systems to allow comparisons and replications. We are also in contact with companies to replicate this validation on their commercial systems. We contacted developers involved in each of the systems of the validation to compute recalls. So far, we received few answers but with enthusiastic interest. Software engineers analysed independently our results for LOG4J, LUCENE, PMD, and QUICKUML, and confirmed the results in Table 3. We gratefully thank M. Adamovic, C. Alphonce, D. Cutting, T. Copeland, P. Gardner, E. Ross, and Y. Shapira for their kind help. We are in the process of increasing the size of our library of known design defects thanks to their kind help. In addition, we believe important to report the results of the detection to the communities developing the systems.

3 Conclusion

The detection of design defects is important to improve the quality of software systems, to ease their evo-

lution, and thus to reduce the overall cost of software development. However, the manual detection of design defects is tedious and time-consuming.

In previous works [8 ; 12], we presented a systematic method, DECOR, that covers the complete process of specifying design defects, generating automatically detection algorithms, and detecting design defects.

In this paper, we presented a validation of the DECOR method using 4 design defects (Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife) and their detections in 10 reverse-engineered designs from open-source systems (ArgoUML, Azureus, GanttProject, Log4J, Lucene, Nutch, PMD, QuickUML, and two versions of Xerces). We reported the precisions and recalls of the detection algorithms for Xerces v2.7.0 and the precisions for the other systems. We showed that the detection algorithms are reasonably efficient and precise and have a good recall. We concluded on the usefulness of our method. We also applied our detection algorithms on Eclipse v3.1.2, demonstrating its scalability and highlighting the problem of balance among numbers of suspicious classes, precisions, and development context.

We are currently in contact with software engineers working on various open-source and industrial systems to apply and further validate our method. We are also contacting other communities to report the results of our detection algorithms, to compute the precisions and recalls of the results, and to improve our specifications of the design defects. Other future work include: using existing tools to improve the implementation of our method; studying key concepts with formal concept analysis; improving the quality and performance of the source code of the generated detection algorithms; computing the recall on other systems; applying our method to other kinds of defects; performing formal usability tests; comparing quantitatively our method with previous work. As regards this last point, we are currently conducting a study on defect detection tools including several tools such as RevJava, FindBugs, PMD, Hammurapi, or Lint4j to assess our tool against existing tools while a first comparison is available in the Related Work.

References

[1] 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.

[2] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering*, 25(5):675–689, 1999.

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

[4] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[5] 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.

[6] 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*, pages 301–314. ACM Press, October 2004.

[7] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN: 0444002057.

[8] 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*, pages 201–216. Hermès Science Publications, March 2006.

[9] Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE — Future of SE Track*, pages 47–60, 2000.

[10] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. In Peter G. Neumann, editor, *Software Engineering Notes*, 17(4):40–52. ACM Press, October 1992.

[11] Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5^{th} edition, November 2001. ISBN: 0-07-249668-1.

[12] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In Sebastian Uchitel and Steve Easterbrook, editors, *Proceedings of the* 21^{st} *conference on Automated Software Engineering*. IEEE Computer Society Press, September 2006. Short paper.

[13] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *proceedings of the* 14^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–56. ACM Press, 1999.