

DECOR : A Tool for the Detection of Design Defects

Naouel Moḥa
Yann-Gaël Guéhéneuc

Ptidej Team – GEODES
Dept of Informatics and Operations Research
University of Montreal, Quebec, Canada
{mohanaou, guehene}@iro.umontreal.ca

ABSTRACT

Software engineers often need to identify design defects, recurring design problems that hinder the development process, to improve and assess the quality of their systems. However, this 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 dedicated language for generating detection algorithms.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.11 [Software Engineering]: Software Architectures—*Domain-specific architectures, Languages (e.g., description, interconnection, definition)*

General Terms

Algorithms, Experimentation, Languages

Keywords

Design defects, antipatterns, code smells, domain-specific language, algorithm generation, detection, Java

1. INTRODUCTION

Quality is an important goal in development process of all applications, from games to life support systems. Quality is usually assessed and improved during formal technical reviews, primarily to detect errors and defects early, before they are passed on to another software activity or released to the customer [7].

We define design defects as solutions to recurring problems that generate negative consequences on the quality of object-oriented systems. 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.

One example of design defects is the Spaghetti Code antipattern, 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 object-orientation mechanisms, such as polymorphism and inheritance, and prevents their use.

The detection of design defects and their early correction substantially reduce the cost of subsequent activities in the development and support phases [7] because designs free of defects are easier to implement, change, and maintain. However, detection in large designs is a time-, resource-consuming, and error-prone activity.

Several approaches and tools have been proposed to detect design defects [5, 6]. However, none of these approaches allow the detection of high-level design defects such as antipatterns. Marinescu [5] presented a metric-based approach to detect code smells with detection strategies, implemented in a tool called IPLASMA. However, metrics cannot express important structural relationships and properties, while in DECOR, detection rules can also be specified using structural and lexical properties and structural relationships. Moreover, the IPLASMA tool was only evaluated in terms of precision; no recall was computed. Tools such as CHECKSTYLE [2], FXCOP [4], and PMD [6] detect problems related to coding standards, bugs patterns or unused code. They focus on implementation problems but do not address higher-level design defects such as antipatterns.

2. DECOR APPROACH

In our approach, DECOR¹, we propose a domain-specific language to specify and generate automatically design defect detection algorithms. A domain-specific language offers greater flexibility than ad hoc algorithms because the domain experts, the software engineers, can specify and modify manually the detection rules using high-level abstractions, taking into account the context, environment, and characteristics of the analysed systems. Moreover, the language allows specifying defect detection algorithms at a high-level of abstraction using key concepts found in their text-based descriptions, not in the underlying ad hoc detection framework, as in previous work.

¹DECOR stands for (*Defect dEtection for CORrection*). However, the correction is out of the scope of this poster, thus we focus on the part of the DECOR method related to detection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.
Copyright 2007 ACM 978-1-59593-882-4/07/0011 ...\$5.00.

Domain-specific Language. The language consists in offering a programmatic means to use key concepts to specify design defects. In particular, it provides the concepts of rules, rule cards which are sets of rules combined using set operators, and properties related to the lexicon, the metrics and the structure of the design. Figure 1 shows the simplified rule card of the Spaghetti Code. This rule card characterises classes as Spaghetti Code using the intersection of rules (lines 2, 3, 6). A class is Spaghetti Code if it declares methods with a very high number of lines of code (line 4), with no parameter (line 5); if it does not use inheritance (line 8), and polymorphism (line 9), and has a name that recalls procedural names (line 11), while declaring/using global variables (line 12).

```

1  RULE_CARD: SpaghettiCode {
2  RULE: SpaghettiCode
   { INTER Inter1 Inter2 };
3  RULE: Inter1
   { INTER LongMethod NoParameter };
4  RULE: LongMethod   { METRIC LOC_METHOD VERY_HIGH 10.0 };
5  RULE: NoParameter  { METRIC MNOPARAM VERY_HIGH 5.0 };
6  RULE: Inter2
   { INTER Inter3 Inter4 };
7  RULE: Inter3
   { INTER NoInheritance NoPolymorphism };
8  RULE: NoInheritance { METRIC DIT 1 0.0 };
9  RULE: NoPolymorphism { STRUCT NO_POLYMORPHISM };
10 RULE: Inter4
   { INTER ProceduralName UseGlobalVariable };
11 RULE: ProceduralName { LEXIC CLASS_NAME
   (Make, Create, Exec...)};
12 RULE: UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
13 };

```

Figure 1: Spaghetti Code Simplified Rule Card.

Generation Process. The generation process consists of three fully automated steps, from the rule cards through their reification to algorithm generation (Figure 2):

1. **Parsing.** The first step consists in parsing the rules cards. A parser is built using JFLEX and JAVACUP² from a grammar, extended with appropriate semantic actions.
2. **Reification.** As a rule card is parsed, the semantic actions produce a model of the rule card, instance of the dedicated SADDL (*Software Architectural Defects Definition Language*) meta-model.
3. **Algorithm Generation.** The generation of the detection algorithms is implemented as visitors on models of rule cards. The generation uses the services of the SAD (*Software Architectural Defects*) framework and is based on templates. Templates are excerpts of Java source with well-defined tags to be replaced by concrete code. We use templates because our previous studies [8, 9] showed that detection algorithms have recurring structures. Thus, we aggregate naturally all common structures of detection algorithms into templates, in contrast to existing approaches, whose algorithms are ad hoc. As we visit the model of a rule card, we replace the tags with the data and values appropriate to the rules. The final source code generated for a

rule card is the detection algorithm of the corresponding design defect and this code is directly executable without any manual interventions.

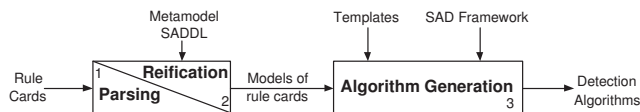


Figure 2: Generation Process.

3. CONCLUSION

In this paper, we introduced an approach for specifying and generating design defect detection algorithms and we illustrated it using the Spaghetti Code design defect. The originality of this approach stems from the definition of a domain-specific language of design defects resulting from an in-depth domain analysis. This language allows the specification and detection of design defects in the form of rule cards, using a set of reusable key concepts resulting from the domain analysis. Using the language, we specified several design defects, generated automatically detection algorithms using templates, and validated the generated detection algorithms in terms of precision and recall, for the first time, on XERCES v2.7.0, an open-source software system. We showed that the detection algorithms are reasonably efficient and precise and have a good recall.

4. REFERENCES

- [1] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998.
- [2] CheckStyle, 2004. <http://checkstyle.sourceforge.net>.
- [3] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1st edition, June 1999.
- [4] FXCop, June 2006. <http://www.gotdotnet.com/team/fxcop/>.
- [5] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.
- [6] PMD, June 2002. <http://pmd.sourceforge.net/>.
- [7] R. S. Pressman. *Software Engineering – A Practitioner’s Approach*. McGraw-Hill Higher Education, 5th edition, November 2001.
- [8] Naouel Moha, Y.-G. Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In S. Uchitel and S. Easterbrook, editors, *Proceedings of the 21st Conference on Automated Software Engineering*. IEEE Computer Society Press, September 2006. Short paper.
- [9] Naouel Moha, Duc-Loc Huynh, and Y.-G. Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. In R. Rousseau, editor, *Actes du 12^e colloque Langages et Modèles à Objets*, pages 201–216. Hermès Science Publications, mars 2006.

²More information on JFlex and JavaCUP are available at <http://www2.cs.tum.edu/projects/cup/>.