Automatic Generation of Detection Algorithms for Design Defects

Naouel Moha and Yann-Gaël Guéhéneuc and Pierre Leduc
Ptidej Team, GEODES Lab
Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
{mohanaou,guehene,leducpie}@iro.umontreal.ca

Abstract

Maintenance is recognised as the most difficult and expansive activity of the software development process. Numerous techniques and processes have been proposed to ease the maintenance of software. In particular, several authors published design defects formalising "bad" solutions to recurring design problems (e.g., anti-patterns, code smells). We propose a language and a framework to express design defects synthetically and to generate detection algorithms automatically. We show that this language is sufficient to describe some design defects and to generate detection algorithms, which have a good precision. We validate the generated algorithms on several programs.

1 Introduction

Large object-oriented programs are expensive to maintain because of bad design practices and architectural drift [8], which make adding, debugging, and evolving features difficult. We define "design defects" as the embodiment of bad design practices in the source code of programs. Informally, design defects are "bad" solutions to recurring design problems that hinder maintenance by decreasing software quality. They encompass problems with different granularities, from architectural problems, such as antipatterns [3], to low-level problems, such as code smells [4] (long methods, long parameter lists, or large classes). The Blob is a typical example of a high-level design defect. A Blob [3] (or God class [10]) corresponds to a large controller class that monopolises most of the processing done by a program. This controller class declares many fields and methods with a low cohesion among one another. It is associated with several simple data classes. It is dependent on the data stored in the data classes. A data class contains only data and accessor methods.

Tools such as SmallLint [2] or PMD [9] detect defects such as bugs, unused code, or syntax errors. These tools focus on code-level problems but do not address higher-level defects because the specification and detection of design defects are tedious and labour-intensive activities. Some authors proposed definitions of design defects [4, 10] and algorithms to detect design defects in programs [5, 7, 11], yet these definitions are informal and subject to misinterpretations and the detection techniques are mostly limited to metrics, such as previous in works by Marinescu [5] or Munro [7].

We attempt to overcome the limitations of previous work on the detection of design defects by defining these defects synthetically and by generating detection algorithms automatically. We enhance the specification and detection of design defects with structural and semantic properties because metrics are not sufficient to detect design defects precisely. Using a method to describe design defects [6], we build the first language to specify design defects in terms of their basic characteristics, such as metrics, structural relationships, and semantic and structural properties. We use this language to specify design defects from the literature. Then, using this language and our framework dedicated to the analysis of programs, we generate detection algorithms automatically from the specifications of design defects. The language is sufficient to describe common design defects and to generate accurate detection algorithms. We validate the generated algorithms on several programs to demonstrate their precision.

The remainder of this paper is organised as follows. Section 2 surveys related work on the specification and detection of design defects. Section 3 presents the language to specify design defects. Section 4 describes the framework to generate detection algorithms for design defects automatically. Section 5 presents case studies of the generated algorithms on different programs. Finally, section 6 concludes and sketches future work.

2 Related Work

In their pioneering work, Fowler and Beck describe 22 code smells, that are low-level design defects in the source code of a program suggesting that maintainers should apply refactorings [4]. They define code

smells in an informal style and propose methods to locate different code smells in source code manually, using code inspection and human judgment. Brown *et al.* [3] published the first book on antipatterns, which are described textually. These books provide inbreadth views on heuristics, code smells, and antipatterns aimed at a wide audience for educational purposes rather than in-depth technical studies. Therefore, it is difficult to build detection algorithms from the textual descriptions directly because they lack precision and are prone to misinterpretations. We defined in a previous work [6] a systematic method to specify design defects from their textual descriptions and generate automatically detection algorithms.

Marinescu [5] presents an approach based on metrics for detecting code smells with *detection strategies*. A detection strategy is a generic algorithm for computing metric values on a source code model and for capturing deviations from "good" design principles using heuristics. The detection algorithm of each code smell is built automatically, but only using set operators on sets of classes with some manually-specified metric values (absolute or relative), which cannot express structural or semantic properties. Munro [7] builds on this previous work and characterises code smells less informally using a generic template and software metrics. Munro and Marinescu's approaches have two important limitations. First, they focus only on code smells and do not apply their detection strategies to higher-level design defects, such as antipatterns. Indeed, it is our understanding that they cannot combine their heuristics easily. Marinescu illustrates his approach with the Blob for only high-level design defect. Second, they use heuristics based on metrics only, which are insufficient to detect design defects precisely, because metrics cannot express important structural and semantic properties. To ease the comparison previous work, we illustrate our approach with the Blob and three other high-level design defects: Functional Decomposition, Spaghetti Code, and Swiss Army Knife.

We improve previous work on the specification and detection of design defects. Based on our systematic method [6], we introduce a language to describe design defects synthetically. This language uses structural relationships, the semantics of names, and structural properties in addition to metrics to characterise design defects, because metrics cannot characterise adequately several properties of high-level design defects. We generate detection algorithms automatically using this language. We detail in the following the specification of design defects using the example of the Blob design defect, the generation of algorithms, and the validation of the algorithms.

3 Specification of Design Defects

We specify design defects synthetically using rule cards, *i.e.*, sets of rules. Rule cards are at the core of our method to generate detection algorithms for design defects automatically.

Rule Card and Rules. A rule card describes a design defect, its code smells and the relationships among code smells. We formalise rule cards with a BNF grammar, which determines the exact syntax for a language. Figure 1 shows the grammar used to express rule cards. A rule card is identified by the keyword `RULE_CARD`, followed by a name and a set of rules specifying this specific design defect as a set of code smells (Figure 1, line 1). A rule describes a code smell as a list of properties (such as metric, lines 7-16, see in the following for more details), its relationships with other code smells (such as associations, lines 18-22), and–or combination with other code smells, based on available operators (such as union, line 5).

Properties. Properties can be of three different types (metrics, structural, or semantics) and define pairs of identifier–value (lines 8-10). A structural property (line 16) is property verified by a method, an interface, a field, or a parameter. A property based on metrics defines a numerical or an ordinal value for a specific metric (line 12). We can sum or subtract metrics (line 11). Numerical values are used to define thresholds or absolute values, whereas ordinal values are used to define values relative to all the classes of the program under analysis. A semantic property relates to the semantics of a class.

Example. Figure 2 illustrates the grammar with the rule card of the Blob design defect. The Blob design defect is divided in two main code smells: Controller Class and Data Class. These two code smells represent classes tied by an association relationship (`ASSOC`). A Controller Class is the union of two other code smells `LargeClassLowCohesion` and `ControllerClassName`, *i.e.*, classes with large number of methods and attributes (`NM + NA`) (line 11), with low cohesion (`LCOM5`) (line 13), and with specific names identified by keywords (`CLASSNAME` and `METHODNAME`) (line 16-17). A Data Class is a class defining accessors (`Accessor`) with a high cohesion (line 20).

Discussion. The language we propose for specifying rule cards offers a greater flexibility than implementing ad-hoc detection algorithms, because it allows describing design defects at a higher-level of abstraction than previous works, including combining rules for code smells and specifying the relationships, structure, and semantics of the suspicious classes. Moreover, the

```
rule_card      ::=  RULE_CARD: string {list_rules};
list_rules     ::=  rule | list_rules rule
rule           ::=  RULE: string {content_rule};
content_rule   ::=  operator rule rule | list_relations | list_prop
operator       ::=  INTER | UNION | DIFF | INCL | NEG

list_prop  ::=  property | operator property property
property   ::= (METRIC: id_metric, value_ordi)
             | (SEMANTIC: id_sem, value_sem)
             | (STRUCT: id_struct, string)
id_metric  ::= id_metric+id_metric | id_metric-id_metric | string
value_ordi ::=  VERY_HIGH | HIGH | MEDIUM | LOW | NONE | NUMBER
id_sem     ::=  CLASSNAME | METHODNAME | FIELDNAME
value_sem  ::=  {cont_semantic}
cont_sem   ::=  string | string, cont_sem
id_struct  ::=  CLASS | INTERFACE | METHOD | FIELD | PARAMETER

list_relations  ::=  relationship | relationship list_relations
relationship    ::=  name_relation: string FROM: string cardinality
                                           TO: string cardinality
name_relation   ::=  ASSOC | AGGREG | COMPOS
cardinality     ::=  ONE | MANY | ONE_OR_MANY | OPTIONALLY_ONE
```

```
RULE_CARD: BlobCard {

RULE: Blob {ASSOC: associated FROM: ControllerClass ONE
                                TO: DataClass MANY};
RULE: ControllerClass
      {INTER LargeClassLowCohesion ControllerClassName};

RULE: LargeClassLowCohesion
      {INTER LargeClass ClassLowCohesion};

RULE: LargeClass {(METRIC: NM + NA, VERY_HIGH)};

RULE: ClassLowCohesion {(METRIC: LCOM5, VERY_HIGH)} ;

RULE: ClassControllerName {UNION
       (SEMANTIC: CLASSNAME, {System, Manager, Controller})
       (SEMANTIC: METHODNAME, {Process, Control, Command})};

RULE: DataClass
      {INTER (STRUCT: METHOD, Accessor) (METRIC: LCOM5, HIGH)};
};
```

language is straightforward to understand and to handle by a maintainer, because it does not require extensive programming skills.

4 Generation of the Algorithms

We use the language based on rule cards defined in the previous section to generate detection algorithms automatically. The automatic generation ensures the traceability between the specifications of the design defects and their detection.

Underlying Framework. The automated generation of detection algorithms relies on the SAD framework (*Software Architectural Defects*). The SAD framework provides the building blocks (related to the concepts of relationships, operators, properties, and ordinal values) common to all detection algorithms. It provides services to build, to access, to compute metrics, to analyse structural relationships, and to perform semantic and structural analyses on a program model.

The SAD framework includes the PADL meta-model (*Pattern and Abstract-level Description Language*) [1], which allows describing object-oriented programs and the structure of solutions of design patterns. PADL offers a set of constituents (classes, interfaces, methods, fields...) to describe programs and the methods on these constituents required to assess their properties.

Structural and semantic properties can be specified via PADL. Properties based on metrics are computed using POM (*Primitives, Operators, Metrics*), an extension to PADL to compute metrics. POM provides several dozen metrics, such as LCOM5 (Lack Of Cohesion in Methods), NM (Number of Methods), and CBO (Coupling Between Object).

The SAD framework also includes the SADDL meta-model (*Software Architectural Defects Definition Language*), which extends the PADL meta-model with constituents related to design defects (identified as key concepts, such as constituents to specify design defects and their code smells) to describe models of rule cards.

Finally, the SAD framework includes algorithms to visit models of rule cards and to generate detection algorithms from these models.

Concrete Generation. The generation of the detection algorithms divides in three steps. The first step consists in generating a parser for the grammar of the rule cards. The second step consists in parsing the rule cards to check if they conform to the BNF grammar. As we parse the rule card, we build a model representing this rule card using the SAD framework, which provides all the constituents required to model rule cards. In the last step, we visit the model of the rule card and generate the detection algorithm corresponding to the rules, using the SAD framework.

5 Validation of the Algorithms

We perform a study on the use of the language and of the SAD framework to express design defects and to detect these design defects in programs.

Our objective is to show through case studies that our language has enough expressive power to describe several design defects of different nature and that the precision of the generated detection algorithms is reasonable, *i.e.*, at least two-third of the results of the generated algorithms are true positive. The precision compares the number of *true* design defects among all detected design defects.

Concretely, we use our language to describe 4 well-known but different design defects described in Brown's book [3]: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. These

design defects include in their descriptions different code smells described in Fowler's book [4]. We studied more that 15 design defects and report here 4 of them. 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/`. We summarise in the following each design defect.

Functional Decomposition. The Functional Decomposition design defect occurs when experienced procedural developers with little knowledge of object-oriented programming implement an object-oriented program. It divides in a main class (a class with a procedural names, such as `Compute` or `Display`, in which inheritance and polymorphism are scarcely used) associated with many data classes (like in the Blob design defect) or small classes (classes with a very low number of methods and fields).

Spaghetti Code. The Spaghetti Code design defect results from procedural thinking in object-oriented programming. It characterises classes with long methods, with methods with no parameters, with no inheritance, defining global variables.

Swiss Army Knife. The Swiss Army Knife design defect is a complex class that offers a high number of services to address many different needs. Utility classes are typical examples of Swiss Army Knives. A Swiss Army Knife is a complex class implementing a high number of interfaces.

Results. We then generate the associated detection algorithms using our SAD framework. We apply these algorithms on 6 open-source programs: Log4J, Lucene, PMD, QuickUML, Xerces v1.0.1, and Xerces v2.7.0. In contrast with previous works, we use *freely available* programs to ease comparisons and replications of this study. We record the times of detection and the number of suspicious classes. We perform two separate manual code inspections to validate the results and compute their precisions. Validation was performed manually because only maintainers can assess the presence of defects in the design depending on their design choices and on the context.

Our language has enough expressive power to describe 15 design defects of different nature. The generated algorithms provide good results in terms of number of suspicious classes and their precisions. The overall precision is reasonable, with 64% or more, in particular for large programs, such as PMD and Xerces. Computation times vary from 1 millisecond for the Swiss Army Knife (the most simple of the 4 design defects) for QuickUML to 2,364 milliseconds for the Spaghetti Code for Xerces v2.7.0. These results confirm the interest of specifying design defects using a defined language, based on metrics as well as structural relationships, structural and semantic properties.

6 Conclusion and Future Work

Maintenance of programs is a tedious and time-consuming activity, mainly because of the quality of programs. We defined a simple language based on a BNF grammar to specify the design defects in terms of metrics and structural and semantic properties. This language describes rule cards, which allows maintainers to specify easily and freely the design defects they are looking for exactly. Then, we presented a framework to support the automatic generation of detection algorithms from the rule cards. This framework provides all constituents and the services to model rule cards and to generate detection algorithms. Finally, we validated the generated algorithms on several open-source programs and reported their precisions. We showed that the detection algorithms are efficient in time and precise. We are currently developing our method (1) to compare in details with previous work; (2) to apply our algorithms on larger programs and to analyse the results of the detection algorithms; and (3) to develop the language and the generation algorithms.

References

[1] H. Albin-Amiot, P. Cointe, and Y.-G. Guéhéneuc. Un métamodèle pour coupler application et détection des design patterns. In *actes du* 8^e *colloque Langages et Modèles à Objets*, volume 8, numéro 1-2/2002 of *RSTI – L'objet*, pages 41–58. Hermès Science Publications, janvier 2002.

[2] J. Brant. Smalllint, April 1997. `http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html`.

[3] 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, 1^{st} edition, March 1998.

[4] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999.

[5] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.

[6] N. Moha, D.-L. Huynh, and Y.-G. Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. In *actes du* 12^e *colloque Langages et Modèles à Objets*, pages 201–216. Hermès Science Publications, March 2006.

[7] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[8] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *Software Engineering Notes*, 17(4):40–52, October 1992.

[9] PMD, June 2002. `http://pmd.sourceforge.net/`.

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

[11] A. Trifu and I. Dragos. Strategy-based elimination of design flaws in object-oriented systems. In *proceedings of the* 4^{th} *international Workshop on Object-Oriented Reengineering*. Universiteit Antwerpen, July 2003.