A Domain Analysis to Specify Design Defects and Generate Detection Algorithms

Naouel Moha[1,2], Yann-Gaël Guéhéneuc[1],
Anne-Françoise Le Meur[2], and Laurence Duchien[2]

[1] Ptidej Team – GEODES, DIRO
University of Montreal, Quebec, Canada
{mohanaou,guehene}@iro.umontreal.ca
[2] Adam Team – INRIA Futurs, LIFL
Université des Sciences et Technologies de Lille, France
{Laurence.Duchien,Anne-Francoise.Le-Meur}@lifl.fr

Abstract. Quality experts often need to identify in software systems design defects, which are recurring design problems, that hinder development and maintenance. Consequently, several defect detection approaches and tools have been proposed in the literature. However, we are not aware of any approach that defines and reifies the process of generating detection algorithms from the existing textual descriptions of defects. In this paper, we introduce an approach to automate the generation of detection algorithms from specifications written using a domain-specific language. The domain-specific is defined from a thorough domain analysis. We specify several design defects, generate automatically detection algorithms using templates, and validate the generated detection algorithms in terms of precision and recall on Xerces v2.7.0, an open-source object-oriented system.

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

1 Introduction

Software quality is an important goal of software engineering because software systems are pervasive and realise vital functions in our societies. It is assessed and improved mainly by quality experts during formal technical reviews, which objective is to detect errors and defects early, before they are passed on to subsequent software engineering activities or released to customers [23].

During the reviews, the experts track design defects, which are "bad" solutions to recurring design problems in object-oriented systems. Design defects are problems resulting from bad design practices [21]. They include problems ranging from high-level and design problems, such as antipatterns [3], to low-level or local problems, such as code smells [9]. They make adding, debugging, and evolving of features difficult. These defects are at a higher-level than Halstead or Fenton's defects, which are "deviations from specifications or expectations

J. Fiadeiro and P. Inverardi (Eds.): FASE 2008, LNCS 4961, pp. 276–291, 2008.

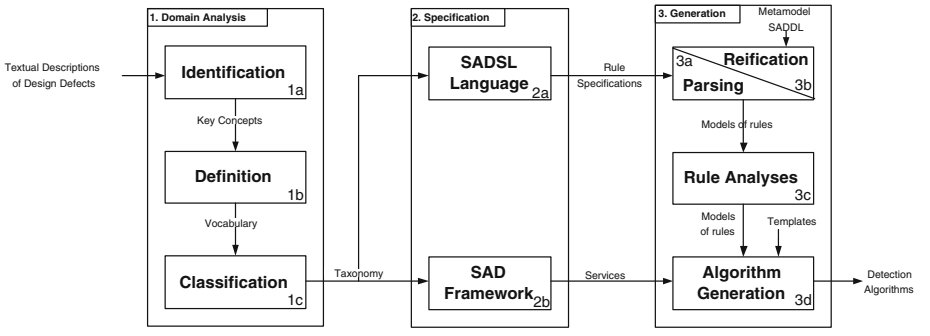

Fig. 1. Generation Process

which might lead to failures in operation" [8,15]. Code smells are symptoms of antipatterns.

Several approaches have been proposed in the literature to specify and detect design defects, for example [16,20,28]. Although showing interesting precisions, these approaches are, to the best of our knowledge, all based on detection algorithms that are defined programmatically and implemented by hand: (1) The detection algorithms are defined at the code level by developers rather than at the domain level by quality experts; (2) The implementation of the algorithms is guided by the services of the underlying detection framework rather than by a study of the textual descriptions of the defects. Thus, it is difficult for quality experts to evaluate the choices made by the developers, to adapt the algorithms to different contexts, and to compare the results of different implementations.

In this paper, we follow the principle of domain analysis to propose a domain-specific language to specify design defects at the domain level and generate automatically detection algorithms from these specifications. Our domain-specific language, SADSL (*Software Architectural Defects Specification Language*), offers the following benefits with respect to previous work: (1) The language is based on the key concepts found in the textual descriptions of the defects rather than on the underlying detection framework; (2) The specifications are used to generate automatically detection algorithms rather than implementing algorithms by hand. Thus, quality experts can specify defects using domain-related abstractions, taking into account the context and characteristics of the analysed systems, and generate traceable detection algorithms.

The new aspects presented in this paper compared with our previous work [18,19] are threefold. First, we present a domain analysis of the key concepts defining design defects and its resulted and enhanced domain-specific language to specify defects, SADSL. However, we re-present the underlying detection framework, SAD (*Software Architectural Defects*) for the sack of clarity. Second, we propose a explicit process for generating detection algorithms automatically using templates. Third, we present the validation of this process including the first study of both precision and recall on a large open-source software system, Xerces v2.7.0. Figure 1 relates the different contributions.

Table 1. List of Design Defects

The *Blob* (called also God class [25]) 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 system, takes most of the decisions, and closely directs the processing of other classes [30]. We identify controller classes using suspicious names such as 'Process', 'Control', 'Manage', 'System', and so on. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.
The *Functional Decomposition* antipattern may occur if experienced procedural developers with little knowledge of object-orientation implement an object-oriented system. Brown describes this antipattern as "a 'main' routine that calls numerous subroutines". The Functional Decomposition design defect consists of a main class, *i.e.*, a class with a procedural name, such as 'Compute' or 'Display', in which inheritance and polymorphism are scarcely used, that is ***associated with small classes***, which declare many private fields and implement only ***few*** methods.
The *Spaghetti Code* is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring ***long methods*** with ***no parameters***, and utilising ***global variables*** for processing. ***Names of classes and methods*** may suggest ***procedural*** programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, ***polymorphism*** and ***inheritance***.
The *Swiss Army Knife* refers to a tool fulfilling a wide range of needs. The Swiss Army Knife design defect is a complex class that offers a high number of services, for example, a complex class implementing a high number of interfaces. A Swiss Army Knife is different from a Blob, because it exposes a high complexity to address all foreseeable needs of a part of a system, whereas the Blob is a singleton monopolising all processing and data of a system. Thus, several Swiss Army Knives may exist in a system, for examples utility classes typically.

In the rest of this paper, Section 2 presents the domain analysis performed on the literature pertaining to design defects. Section 3 presents SADSL and SAD. Section 4 describes the generation process of detection algorithms. Section 5 validates our contributions with the specification and detection of four design defects: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, on the open-source system Xerces v2.7.0. Section 6 surveys related work. Section 7 concludes and presents future work.

2 Domain Analysis of Design Defects

"Domain analysis is a process by which information used in developing software systems is identified, captured, and organised with the purpose of making it reusable when creating new systems" [24].

In the context of design defects, *information* relates to the defects, *software systems* are detection algorithms, and the information on design defects must be *reusable* when specifying new design defects. Thus, we have studied the textual descriptions of design defects in the literature to identify, define, and organise the key concepts of the domain, Steps 1a–1c in Figure 1.

2.1 Identification of the Key Concepts

The first step of the domain analysis consists of reviewing the literature on design defects, in particular books and articles, for example these cited in the related work in Section 6, to identify essential key concepts. This step is performed manually so its description would be necessarily narrative. Therefore, to illustrate

this step, we use the example of the Spaghetti Code antipattern. We summarise the textual description of the Spaghetti Code [3, page 119] in Table 1 along with these of the Blob [3, page 73], Functional Decomposition [3, page 97], and Swiss Army Knife [3, page 197].

In the textual description of the Spaghetti Code, we identify the key concepts (highlighted) of classes with long methods, procedural names and with methods with no parameter, of classes defining global variables, and of classes not using inheritance and polymorphism.

We perform this first step iteratively: for each description of a defect, we extract all key concepts, compare them with existing concepts, and add them to the set of key concepts avoiding synonyms, a same concept with two different names, and homonyms, two different concepts with the same name. We study 29 defects, which included 8 antipatterns and 21 code smells. These 29 defects are representative of the whole set of defects described in the literature and include about 60 key concepts.

2.2 Definition of the Key Concepts

The key concepts include metric-based heuristics as well as structural and lexical information. In the second step, we define the key concepts precisely and form a unified vocabulary of reusable concepts to describe defects. For lack of space, we cannot present the definitions of each key concept but introduce a classification of the key concepts according to the types of properties on which they apply: measurable, lexical, and structural properties.

Measurable properties pertain to concepts expressed with measures of internal attributes of the constituents of systems (classes, interfaces, methods, fields, relationships, and so on). A measurable property defines a numerical or an ordinal value for a specific metric. Ordinal values are defined with a 5-point Likert scale: very high, high, medium, low, very low. Numerical values are used to define thresholds whereas ordinal values are used to define values relative to all the classes of a system under analysis.

These properties also related to a set of metrics identified during the domain analysis, including Chidamber and Kemerer metric suite [6]: depth of inheritance `DIT`, lines of code in a class `LOC_CLASS`, lines of code in a method `LOC_METHOD`, number of attributes declared in a class `NAD`, number of methods declared in a class `NMD`, lack of cohesion in methods `LCOM`, number of accessors `NACC`, number of private fields `NPRIVFIELD`, number of interfaces `NINTERF`, number of methods with no parameters `NMNOPARAM`.

Lexical Properties relate to the concepts pertaining to the vocabulary used to name constituents. They characterise constituents with specific names, defined in a list of keywords. In future work, we plan to use the WordNet lexical database of English to deal with synonyms.

Structural Properties pertain to concepts related to the structure of the constituents of systems. For example, the property `USE_GLOBAL_VARIABLE` is used to check if a class uses global variables, and the property `NO_POLYMORPHISM` to check if a class does not/prevents the use of polymorphism. System classes and

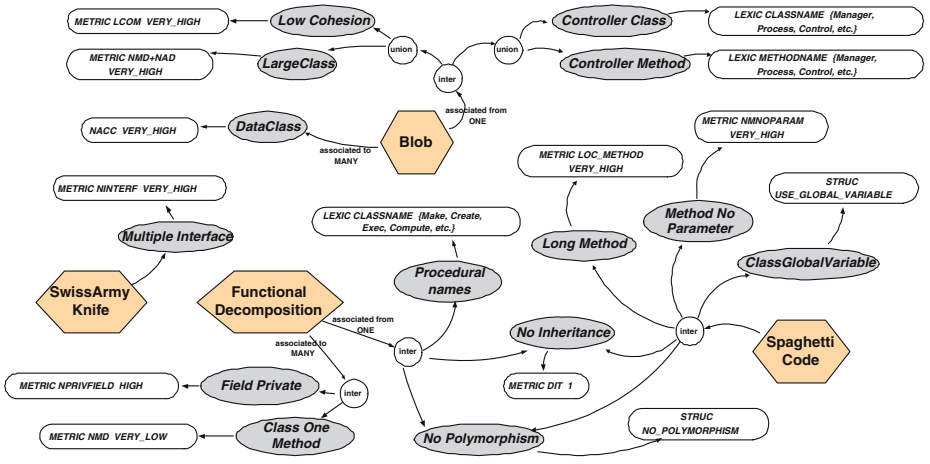

Fig. 2. Taxonomy of Design Defects. (Hexagons are antipatterns, gray ovals are code smells, white ovals are properties, and circles are set operators.)

interfaces characterised by the previous properties may be, in addition, linked with one another with three types of relationships: association, aggregation, and composition. Cardinalities define the minimum and the maximum numbers of instances of rules that participate in a relationship.

In the example of the Spaghetti Code, we obtain the following classification: measurable properties include the concepts of *long methods*, *methods with no parameter*, *inheritance*; lexical properties include the concepts of *procedural names*; structural properties include the concepts of *global variables*, *polymorphism*. Structural properties and relationships among constituents appear in the Blob and Functional Decomposition (see the key concepts *depends on data* and *associated with small classes*).

We also observe during the domain analysis that properties can be combined using **set operators**, such as intersection and union. For example, all properties must be present to characterise a class as Spaghetti Code.

2.3 Classification of the Key Concepts

Using the key concepts and their definitions, we build a taxonomy of defects by using all relevant key concepts to relate code smells and antipatterns on a single map and clearly identify their relationships.

We produce a map organising consistently defects and key concepts. This map is important to prevent misinterpretation by clarifying and classifying defects. It is similar in purpose to Gamma *et al*'s Pattern Map [12, inside back cover]. For lack of space, we only show in Figure 2 the taxonomy from the domain analysis of the four design defects in Table 1.

The taxonomy shows the structural relationships or set combinations among antipatterns (hexagons) and code smells (ovals in gray), and their relation with

```
1  CODESMELL   define  LongMethod        as  METRIC LOC_METHOD with   VERY_HIGH and 10.0;
2  CODESMELL   define  NoParameter       as  METRIC NMNOPARAM with   VERY_HIGH and 5.0;
3  CODESMELL   define  NoInheritance     as  METRIC DIT       with   1 and 0.0;
4  CODESMELL   define  NoPolymorphism    as  STRUC NO_POLYMORPHISM;
5  CODESMELL   define  ProceduralName    as  LEXIC  CLASS_NAME with   (Make, Create, Exec);
6  CODESMELL   define  UseGlobalVariable as  STRUC USE_GLOBAL_VARIABLE;
7  CODESMELL   define  ClassOneMethod    as  METRIC NMD       with   VERY_LOW and 10.0;
8  CODESMELL   define  FieldPrivate      as  METRIC NPRIVFIELDwith   HIGH and 10.0;
9  ANTIPATTERN  define  SpaghettiCode    as  {
             ((LongMethod INTER NoParameter) INTER (NoInheritance INTER NoPolymorphism))
          INTER
             (ProceduralName INTER UseGlobalVariable)};
10 ANTIPATTERN  define  FunctionalDecomposition as  {
             ASSOC FROM (ProceduralName INTER (NoInheritance INTER NoPolymorphism)) ONE
             TO (ClassOneMethod UNION FieldPrivate) MANY};
```

Fig. 3. Specifications of the Spaghetti Code and Functional Decomposition

measurable, structural, and lexical properties (ovals in white). It gives an overview of all key concepts that characterise the four defects and differentiates the key concepts as either structural relationships between code smells or their properties (measurable, structural, lexical). It also makes explicit the relationships among high- and low-level defects.

3 Specification of Design Defects

The domain analysis performed in the previous section provides a set of key concepts, their definition, and a classification. Then, following Steps 2a and 2b in Figure 1, we design SADSL, a domain-specific language (DSL) [17] to specify defects in terms of their measurable, structural and lexical properties, and SAD, a framework providing the services required to make operational computation of the properties, set operations, and so on.

We build SADSL and SAD using and only using the identified key concepts, thus they both capture domain expertise. Therefore, SADSL and SAD differ from general purpose languages and their runtime environments, which are designed to be universal [7]. They also differ from previous work on design defect detection, which did not define *explicitly* a DSL and in which the detection framework drove the specification of the detection algorithms.

Thus, with SADSL and SAD, it is easier for quality experts to understand design defect specifications and to specify new defects because these are expressed using domain-related abstractions and focus on *what* to detect instead of *how* to detect it [7].

3.1 SADSL

With SADSL, quality experts specify defects as sets of rules. Rules are expressed using key concepts. They can specify code smells, which correspond to measurable, lexical, or structural properties, or express antipatterns, which correspond

```
1  rulespec          ::= (codesmell)+ (antipattern)+
2  codesmell         ::= CODESMELL   define codesmellName   as  csContent ;
3  antipattern       ::= ANTIPATTERN define antipatternName as  { apContent };

4  csContent         ::= property
5  apContent         ::= codesmellName | (apContent operator apContent) | relationship
6  operator          ::= INTER | UNION | DIFF

7  property          ::= METRIC metricID with  metricValue and fuzziness
8                        |LEXIC lexicID   with  ((lexicValue,)+)
9                        |STRUC structID
10 metricID          ::= DIT | NINTERF | NMNOPARAM | LCOM | LOC_METHOD
11                       |LOC_CLASS | NAD | NMD | NACC | NPRIVFIELD
12                       |metricID + metricID
13                       |metricID - metricID
14 metricValue       ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW | NUMBER
15 lexicID           ::= CLASS_NAME | INTERFACE_NAME | METHOD_NAME | FIELD_NAME | PARAMETER_NAME
16 structID          ::= USE_GLOBAL_VARIABLE | NO_POLYMORPHISM
17                       |ABSTRACT_CLASS | IS_DATACLASS | ACCESSOR_METHOD
18                       |FUNCTION_CLASS | FUNCTION_METHOD | STATIC_METHOD
19                       |PROTECTED_METHOD | OVERRIDDEN_METHOD
20                       |INHERITED_METHOD | INHERITED_VARIABLE

21 relationship      ::= relationshipType FROM apContent cardinality TO apContent cardinality
22 relationshipType  ::= ASSOC | AGGREG | COMPOS
23 cardinality       ::= ONE | MANY | ONE_OR_MANY

24 codesmellName, antipatternName, lexicValue ∈ string
25 fuzziness ∈ double { 0...100 }
```

Fig. 4. BNF Grammar of Design Defect Rule Specifications

to combinations of code smells using set operators or structural relationships. Figure 3 shows the specifications of the Spaghetti Code and Functional Decomposition antipatterns and their code smells. These two antipatterns shared some code smells as shown in the taxonomy in Figure 2.

A Spaghetti Code is specified using the intersection of two rules, which are intersections of two other rules (line 9). A class is Spaghetti Code if it declares methods with a very high number of lines of code (measurable property, line 1), with no parameter (measurable property, line 2); if it does not use inheritance (measurable property, line 3), and polymorphism (structural property, line 4), and has a name that recalls procedural names (lexical property, line 5), while declaring/using global variables (structural property, line 6). The float value after the keyword '`and`' in measurable properties corresponds to the degree of fuzziness, which is the margin acceptable in percentage around the numerical value (line 3) or around the threshold relative to the ordinal value (lines 1-2). We further explain the ordinal values and the fuzziness in Section 4.

We formalise specifications with a Backus-Naur Form (BNF) grammar, shown in Figure 4. A specification lists first a set of code smells and then a set of antipatterns (Figure 4, line 1). A code smell is defined as a property. A property can be of three different kinds: measurable, structural, or lexical, and define pairs of identifier–value (lines 7–9). The BNF grammar specifies only a subset of possible structural properties, other can be added as new domain analyses are performed. The antipatterns are combinations of the code smells defined in the specification using set operators (lines 5–6). The antipatterns can also be linked

to these code smells using relationships such as the composition, aggregation or association(lines 5, 21–23). Each code smell is identified by a name (line 2) and can be reused in the definition of antipatterns. Thus, the specification of the code smells, which can be viewed as a repository of code smells, can be reused or modified for the specification of different antipatterns.

3.2 SAD

The SAD framework provides services that implement operations on the relationships, operators, properties, and ordinal values. It also provides services to build, access, and analyse systems. Thus, with SAD, it is possible to compute metrics, analyse structural relationships, perform lexical and structural analyses on classes, and apply the rules. The set of services and the overall design of the framework are directed by the key concepts and the domain analysis. SAD represents a super-set of previous detection frameworks and therefore could delegate part of its services to exiting frameworks.

SAD is built upon the PADL meta-model (*Pattern and Abstract-level Description Language*), a language-independent meta-model to represent object-oriented systems, including binary class relationships [13] and accessors, and on the POM framework (*Primitives, Operators, Metrics*) for metric computation [14]. PADL offers a set of constituents from which we can build models of systems. It also offers methods to manipulate these models easily and generate other models, using the Visitor design pattern. We choose PADL because it is mature with 6 years of active development and is maintained in-house.

3.3 Discussions

The defect specifications are self-documenting and express naturally the textual descriptions of the defects. They can be modified by quality experts either to refine them by adding new rules or modifying existing ones to take into account the contexts of the systems. For example, in small applications, a domain expert could consider as defects classes with a high `DIT` but not in large systems. In a management application, a domain expert could also consider different keywords as indicating controller classes. Thus, specifications can be modified easily at the domain level without any knowledge of the underlying detection framework.

4 From Specifications to Detection Algorithms

The problem that we solve is the automatic transition from the specifications to detection algorithms to avoid the manual implementation of algorithms, which is costly and not reusable, and to ensure the traceability between specifications and occurrences of defects detected. Thus, the automatic generation of detection algorithms spares the experts or developers of implementing by hand the detection algorithms and allows them to save time and resources.

The generation process consists of four fully automated steps, starting from the specifications through their reification to algorithm generation, as shown in Figure 1, Steps 3a–3d, and detailed below.

4.1 Parsing and Reification

The first step consists of parsing the design defect specifications. A parser is built using JFlex and JavaCUP (cf. `http://www2.cs.tum.edu/projects/cup/`) from the BNF grammar, extended with appropriate semantic actions.

Then, as a specification is parsed, the second step consists of reifying the specifications based on the dedicated SADDL meta-model (*Software Architectural Defects Definition Language*). The meta-model is a representation of the abstract syntax tree generated by the parser. The meta-model SADDL defines constituents to represent specifications, rules, set operators, relationships among rules, and properties. The result of this reification is a SADDL model of the specification, instance of class `Specification` defined in SADDL. An instance of `Specification` is composed of objects of type `IRule`, which describes rules that can be either simple or composite. A composite rule, `CompositeRule`, is a rule composed of other rules (Composite design pattern). Rules are combined using set operators defined in class `Operators`. Structural relationships are enforced using methods defined in class `Relationships`. The SADDL meta-model also implements the Visitor design pattern.

4.2 Rule Analyses

This step consists of visiting the models of specifications and applying some consistency and domain-specific analyses. These analyses aim to identify incoherent or meaningless rules before generating the detection algorithms. Consistency analyses consist of verifying that specifications are not inconsistent, redundant, or incomplete. An inconsistent specification is, for example, two code smells defined with identical names but different properties. A redundant specification corresponds, for example, to two code smells defined with different names but identical properties. An example of an incomplete specification is a code smell referenced in the rule of an antipattern but not defined in the rule set of code smells. Domain-specific analyses consist of verifying that the rules are conform to the domain. For example, the value associated to a metric has a meaning in the domain : typically, a measurable property with the metric number of methods declared `NMD` equal to a float has no meaning in the domain.

4.3 Algorithm Generation

The generation of the detection algorithms is implemented as a set of visitors on models of specifications. The generation targets the services of the SAD framework and is based on templates. Templates are excerpts of Java code source with well-defined tags to be replaced by concrete code. We use templates because our previous studies [18,19] showed that detection algorithms have recurring structures. Thus, we aggregate naturally all common structures of detection algorithms into templates. As we visit the model of a specification, we replace the tags with the data and values appropriate to the rules. The final source code generated for a specification is the detection algorithm of the corresponding design defect and this code is directly executable without any manual interventions.

We detail in the following the generation of the detection algorithms of the set operators and the measurable properties. We do not present the generation of the lexical properties, structural properties, and structural relationships because they are similar to the ones presented here.

Measurable Properties. The template given in Figure 5(e) is a class called `<CODESMELL>Detection` that extends the class `CodeSmellDetection` and implements `ICodeSmellDetection`. It declares the method `performDetection()`, which consists of computing the specified metric on each class of the system. All the metric values are compared with one another with a boxplot, a statistical technique [4], to identify ordinal values, *i.e.*, outlier or normal values. Then, the boxplot returns only the classes with metric values that verify the ordinal values. Figure 5(c) presents the process of generating code to verify a measurable property defined in the specification of the Spaghetti Code on a set of constituents. When the rule is visited in the model of the specification, we replace the tag `<CODESMELL>` by the name of the rule, *`LongMethod`*, tag `<METRIC>` by the name of the metric, `LOC_METHOD`, tag `<FUZZINESS>` by the value 10.0, and tag `<ORDINAL_VALUES>` by the method associated with the ordinal value `VERY_HIGH`. Figure 5(g) presents the code generated for the rule given in Figure 5(a).

Set Operators. The rules can be combined using set operators such as the intersection (Figure 5(b). The code generation for set operators is quite different from the generation of properties. The template given in Figure 5(f) contains also a class called `<CODESMELL>Detection` that extends the class `CodeSmellDetection` and implements `ICodeSmellDetection`. However, the method `performDetection()` consists of combining with a set operator the list of classes in each operand of the rule given in Figure 5(b) and returning classes that satisfy this combination. Figure 5(d) presents the process related to the code generation for set operators in the specification of the Spaghetti Code. When an operator is visited in the model of the specification, we replace the tags associated to the operands of the rule, `operand1: LongMethod, operand2: NoParameter`, and the tag `<OPERATION>` by the type of set operator specified in the specification, *i.e.*, intersection. The operands correspond to detection classes generated when visiting other rules (Figure 5(h)).

Discussion. The generated algorithms are by construct deterministic. We do not need to revise manually the code because the generation process ensures the correctness of the code source with respect to the specifications. This generated code tends sometimes itself towards Spaghetti Code and could be improved using polymorphism, yet it is automatically generated and is not intended to be read by quality experts and it will be improved in future work. We do not report the generation times because it takes only few seconds.

5 Validation

We validate our contributions by specifying and detecting four design defects and computing the precision and recall of the generated algorithms. We use Xerces

```
CODESMELL define LongMethod as
METRIC LOC_METHOD with VERY_HIGH and 10.0;
```

(a) Excerpt of the Spaghetti Code.

```
ANTIPATTERN define SpaghettiCode as {
    ((LongMethod ∩ NoParameter) ...
```

(b) Excerpt of the Spaghetti Code.

```
public void visit(IMetric aMetric) {
  replaceTAG("<CODESMELL>", aRule.getName());
  replaceTAG("<METRIC>", aMetric.getName());
  replaceTAG(<FUZZINESS>,
    aMetric.getFuzziness());
  replaceTAG(<ORDINAL_VALUE>,
    aMetric.getOrdinalValue());
}
private String getOrdinalValue(int value) {
  switch (value) {
    case VERY_HIGH :
      "getHighOutliers";
    case HIGH :
      "getHighValues";
    case MEDIUM :
      "getNormalValues";
    ...
  }
```

(c) Visitor.

```
public void visit(IOperator anOperator) {
  replaceTAG("<OPERAND1>",
    anOperator.getOperand1());
  replaceTAG("<OPERAND2>",
    anOperator.getOperand2());
  switch (anOperator.getOperatorType()) {
    case OPERATOR_UNION :
      operator = "union";
    case OPERATOR_INTER :
      operator = "intersection";
    ...
  }
  replaceTAG("<OPERATION>", operator);
```

(d) Visitor.

```
public class <CODESMELL>Detection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public Set performDetection() {
    IClass c = iteratorOnClasses.next();
    LOCofSetOfClasses.add(
      Metrics.compute(<METRIC>, c));
    ...
    BoxPlot boxPlot = new BoxPlot(
      <METRIC>ofSetOfClasses, <FUZZINESS>);
    Map setOfOutliers =
      boxPlot.<ORDINAL_VALUE>();
    ...
    suspiciousCodeSmells.add( new CodeSmell(
      <CODESMELL>, setOfOutliers));
    ...
    return suspiciousCodeSmells;
  }
```

(e) Template.

```
public class <CODESMELL>Detection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public void performDetection() {
    ICodeSmellDetection cs<OPERAND1> =
      new <OPERAND1>Detection();
    op1.performDetection();
    Set set<OPERAND1> =
      cs<OPERAND1>.listOfCodeSmells();
    ICodeSmellDetection cs<OPERAND2> =
      new <OPERAND2>Detection();
    op2.performDetection();
    Set set<OPERAND2> =
      cs<OPERAND2>.listOfCodeSmells();
    Set setOperation = Operators.getInstance().
      <OPERATION>(set<OPERAND1>, set<OPERAND2>);
    this.setSetOfSmells(setOperation);
  }
```

(f) Template.

```
public class LongMethodDetection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public Set performDetection() {
    IClass c = iteratorOnClasses.next();
    LOCofSetOfClasses.add(
    Metrics.compute("LOC_METHOD", c));
    ...
    BoxPlot boxPlot = new BoxPlot(
      LOC_METHODofSetOfClasses, 10.0);
    Map setOfOutliers =
      boxPlot.getHighOutliers();
    ...
    suspiciousCodeSmells.add( new CodeSmell(
      LongMethod, setOfOutliers));
    ...
    return suspiciousCodeSmells;
  }
```

(g) Generated Code.

```
public class Inter1
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public void performDetection() {
    ICodeSmellDetection csLongMethod =
      new LongMethodDetection();
    csLongMethod.performDetection();
    Set setLongMethod =
      csLongMethod.listOfCodeSmells();
    ICodeSmellDetection csNoParameter =
      new NoParameterDetection();
    csNoParameter.performDetection();
    Set setNoParameter =
      csNoParameter.listOfCodeSmells();
    Set setOperation = Operators.getInstance().
      intersection(setLongMethod, setNoParameter);
    this.setSetOfSmells(setOperation);
  }
```

(h) Generated Code.

Fig. 5. Code Generation for Measurable Properties (left) and Set Operators (right)

v2.7.0, a framework for building XML parsers in Java. XERCES contains 71,217 lines of code, 513 classes, and 162 interfaces. We seek to obtain a recall of 100% because quality experts need all design defects to improve software quality and ease formal technical reviews. We are not aware of any other work reporting both precision and recall for design defect detection algorithms and will gratefully provide our data for comparison with other work.

5.1 Validation Process

First, we build a model of XERCES. This model is obtained by reverse engineering. Then, we apply the generated detection algorithms on the model of the system and obtain all suspicious classes that have potential design defects. The list of suspicious classes is returned in a file. We validate the results of the detection algorithms by analysing the suspicious classes in the context of the complete model of the system and its environment.

We recast the validation in the domain of information retrieval and use the measures of precision and recall, where precision assesses the number of true identified defects, while recall assesses the number of true defects missed by the algorithms [10]. The computation of precision and recall is performed using independent results obtained manually because only quality experts can assess whether a suspicious class is *indeed* a defect or a false positive, depending on the specifications and the context and characteristics of the system. We asked three master students and two independent software engineers to analyse manually XERCES using only Brown and Fowler's books to identify design defects and compute the precision and recall of the algorithms. Each time a doubt on a candidate class arose, they considered the books as reference in deciding by consensus whether or not this class was actually a design defect. This task is tedious, some design defects may have been missed by mistake, thus we have asked other software engineers to perform this same task to confirm our findings and on other systems to increase our database.

5.2 Results

Table 2 reports detection times, numbers of suspicious classes, and precisions and recalls. 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 system but include accesses to compute metrics and check structural relationships and lexical and structural properties.

The recalls of the generated algorithms are 100% for each 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 quality experts to analyse by hand, with respect to analysing the entire system, 513 classes, manually.

For the Spaghetti Code, we found 76 suspicious classes. Out of these 76 suspicious classes, 46 are indeed Spaghetti Code previously identified in XERCES manually by software engineers independent of the authors, which leads to a precision of 60.53% and a recall of 100.00% (see third line in Table 2). The result file

Table 2. Precision and Recall in XERCES v2.7.0. (In parenthesis, the percentage of classes affected by a design defect.). The number of classes in XERCES v2.7.0 is 513.

Design Defects	Numbers of Known True Positives		Numbers of Detected Defects		Precision	Recall	Time
Blob	39	(7.60%)	44	(8.58%)	88.64%	100.00%	2.45s
Functional Decomp.	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

contains all suspicious classes, including class `org.apache.xerces.impl.xpath.regex.RegularExpression` declaring 57 methods. Among these 57 methods, method `matchCharArray(...)` is typical of Spaghetti Code: it does not use inheritance and polymorphism, uses 18 class variables, and weighs 1,246 LOC.

5.3 Discussion

The validation shows that the specifications of the design defects lead to generated detection algorithms with expected recalls and good precisions. Thus, it confirms that: (1) The language allows describing several design defects. We described four different design defects, composed of 15 code smells; (2) The generated detection algorithms have a recall of 100%, *i.e.*, all known design defects are detected, and an average precision greater than 40%, *i.e.*, the detection algorithms report less than 2/3 of false positives with respect to the number of true positives; and, (3) The complexity of the generated algorithms is reasonable, *i.e.*, the generated algorithms have computation times of few seconds. Computation is fast because the complexity of our detection algorithms depends only on the number of classes in the system, n, and on the number of properties to verify. The complexity of the generated detection algorithms is $(c + op) \times \mathcal{O}(n)$, where c is the number of properties and op of operators.

The results depend on the specifications of the defects. The specifications must be neither too loose, not to detect too many suspicious classes, nor too constraining, and miss design defects. With SADSL, quality experts can refine the specifications of the design defects easily, according to the detected suspicious classes and their knowledge of the system through iterative refinement.

6 Related Work

Several defect detection approaches and tools have been proposed in the literature. We only present approaches and tools that are directly related to design defects. We are not aware of any approach that is based on an explicit domain analysis and the resulting domain-specific language.

Several books provide in-breadth views on pitfalls [29], heuristics [25], code smells [9], and antipatterns [3] aimed at a wide audience for educational purposes. However, they describe *textually* design defects and thus, it is difficult to build detection algorithms from their textual descriptions because they lack precision and are prone to misinterpretation. Travassos *et al.* [27] introduced a process

based on manual inspections to identify design defects. No attempt was made to automate this process and thus it does not scale to large systems easily. Also, it only covers the manual detection of defects, not their specification.

Several semi-automatic approaches for the detection of defects exist. Among these, Marinescu [16] presented a metric-based approach to detect code smells with *detection strategies*, implemented in a tool called iPlasma. This approach introduces metric-based strategies capturing deviations from good design principles. This process is simplistic and does not provide enough details to reproduce the declaration of detection strategies in other tools or to guide the definition of new strategies. However, detection strategies are a step towards precise specifications of code smells. As our tool, iPlasma implements the technique of the boxplot. However, the mapping from the relative values of the boxplot with the metrics is not explicit. In our approach, we not only explicit this technique but also we enhance it with fuzzy logic and, thus, alleviates the problem related to the definition of thresholds. It is difficult to compare our approach with this approach because its detection algorithms are ad hoc and black box. Finally, the iPlasma tool was only evaluated in terms of precision, no recall was computed. Munro [20] also noticed the limitations of the textual descriptions and proposed a template to describe code smells more systematically. It is a step towards more precise specifications of code smells but code smells remain nonetheless textual descriptions subject to misinterpretation. Munro also proposed metric-based heuristics to detect code smells, which are similar to Marinescu's detection strategies. Alikacem and Sahraoui proposed a description language of quality rules to detect violations of quality principles and low-level defects [1].

Tools such as PMD [22], CheckStyle [5], and FXCop [11] detect problems related to coding standards, bugs patterns or unused code. Thus, they focus on implementation problems but do not address higher-level design defects such as antipatterns. Crocopat [2] provides an efficient language to manipulate relations of any arity with a simple and expressive query and manipulation language. However, this language is at a low-level of abstraction and requires quality experts to understand the implementation details of its underlying model.

7 Conclusion and Future Work

In this paper, we introduced a domain analysis of the key concepts defining design defects; a domain-specific language to specify defects, SADSL, and its underlying detection framework, SAD; and a process for generating detection algorithms automatically. We implemented SADSL, SAD and the generation process and studied the precision and recall of four generated algorithms on Xerces v2.7.0. We showed that the detection algorithms are efficient and precise, and have 100% recall. With SADSL, SAD, and the generation process, quality experts can specify defects at the domain level using their expertise, generate detection algorithms, and detect defects during formal technical reviews.

The validation in terms of precision and recall sets a landmark for future quantitative comparisons. Thus, we plan to perform such a comparison of our

work with previous approaches. We also plan to integrate the WordNet lexical database of English into SAD, to improve the generated code in terms of quality and reusability, and to compute the precision and recall on more systems. We will assess the flexibility of the code generation offered when using XML technologies [26]. We will also perform usability studies of the language with quality experts.

Acknowledgments. We thank Giuliano Antoniol, Kim Mens, Dave Thomas, and Stéphane Vaucher for fruitful discussions. We also thank the master students and software engineers who performed the manual analysis and Duc-Loc Huynh and Pierre Leduc for their help in building SAD.

References

1. Alikacem, E.H., Sahraoui, H.: Détection d'anomalies utilisant un langage de description de règle de qualité. In: actes du 12^e colloque LMO, pp. 185–200 (2006)
2. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. Transactions on Software Engineering 31(2), 137–149 (2005)
3. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick III, H.W., Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edn (1998)
4. Chambers, J.M., Clevelmd, W.S., Kleiner, B., Tukey, P.A.: Graphical methods for data analysis (1983)
5. CheckStyle (2004), `http://checkstyle.sourceforge.net`
6. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
7. Consel, C., Marlet, R.: Architecturing software using: A methodology for language development. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 170–194. Springer, Heidelberg (1998)
8. Fenton, N.E., Neil, M.: A critique of software defect prediction models. Software Engineering 25(5), 675–689 (1999)
9. Fowler, M.: Refactoring – Improving the Design of Existing Code, 1st edn. Addison-Wesley, Reading (1999)
10. Frakes, W.B., Baeza-Yates, R.: Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs (1992)
11. FXCop (2006), `http://www.gotdotnet.com/team/fxcop/`
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley, Reading (1994)
13. Guéhéneuc, Y.-G., Albin-Amiot, H.: Recovering binary class relationships: Putting icing on the UML cake. In: Proceedings of the 19^{th} OOSPLA Conference, pp. 301–314 (2004)
14. Guéhéneuc, Y.-G., Sahraoui, H., Zaidi, F.: Fingerprinting design patterns. In: Proceedings of the 11^{th} WCRE Conference, pp. 172–181 (2004)
15. Halstead, M.H.: Elements of Software Science. Operating and programming systems series. Elsevier Science Inc., New York (1977)
16. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20^{th} ICSM Conference, pp. 350–359 (2004)
17. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)

18. Moha, N., Guéhéneuc, Y.-G., Leduc, P.: Automatic generation of detection algorithms for design defects. In: Proceedings of the 21^{st} ASE Conference (2006)
19. Moha, N., Huynh, D.-L., Guéhéneuc, Y.-G.: Une taxonomie et un métamodèle pour la détection des défauts de conception. In: actes du 12^{e} colloque LMO, pp. 201–216 (2006)
20. Munro, M.J.: Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Proceedings of the 11^{th} Metrics Symposium (2005)
21. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. Software Engineering Notes 17(4), 40–52 (1992)
22. PMD (2002), `http://pmd.sourceforge.net/`
23. Pressman, R.S.: Software Engineering – A Practitioner's Approach, 5th edn. McGraw-Hill Higher Education (2001)
24. Prieto-Díaz, R.: Domain analysis: An introduction. Software Engineering Notes 15(2), 47–54 (1990)
25. Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Reading (1996)
26. Swint, G.S., Pu, C., Jung, G., Yan, W., Koh, Y., Wu, Q., Consel, C., Sahai, A., Moriyama, K.: Clearwater: extensible, flexible, modular code generation. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 144–153 (2005)
27. Travassos, G., Shull, F., Fredericks, M., Basili, V.R.: Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: Proceedings of the 14^{th} OOSPLA Conference, pp. 47–56 (1999)
28. Trifu, A., Marinescu, R.: Diagnosing design problems in object oriented systems. In: Proceedings of the 12^{th} WCRE Conference (2005)
29. Webster, B.F.: Pitfalls of Object Oriented Development. M & T Books (1995)
30. Wirfs–Brock, R., McKean, A.: Object Design: Roles, Responsibilities and Collaborations. Addison-Wesley Professional, Reading (2002)