From a Domain Analysis to the Specification and Detection of Code and Design Smells

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

[1] Ptidej Team – GEODES, DIRO
University of Montreal, Quebec, Canada
{mohanaou,guehene}@iro.umontreal.ca
[2] Adam Team – INRIA Lille, LIFL CNRS UMR 8022
Université des Sciences et Technologies de Lille, France
{Laurence.Duchien,Anne-Francoise.Le-Meur,Alban.Tiberghien}@lifl.fr

Abstract. Code and design smells are recurring design problems in software systems that must be identified to avoid their possible negative consequences on development and maintenance. Consequently, several smell detection approaches and tools have been proposed in the literature. However, so far, they allow the detection of predefined smells but the detection of new smells or smells adapted to the context of the analysed systems is possible only by implementing new detection algorithms manually. Moreover, previous approaches do not explain the transition from specifications of smells to their detection. Finally, the validation of the existing approaches and tools has been limited on few proprietary systems and on a reduced number of smells. In this paper, we introduce an approach to automate the generation of detection algorithms from specifications written using a domain-specific language. This language is defined from a thorough domain analysis. It allows the specification of smells using high-level domain-related abstractions. It allows the adaptation of the specifications of smells to the context of the analysed systems. We specify 10 smells, generate automatically their detection algorithms using templates, and validate the algorithms in terms of precision and recall on Xerces v2.7.0 and GanttProject v1.10.2, two open-source object-oriented systems. We also compare the detection results with those of a previous approach, iPlasma.

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

1 Introduction

Following their development and delivery to the clients, software systems enter the maintenance phase. During this phase, systems are modified and adapted to support new user requirements and changing environments, which can lead to the introduction of "bad smells" in the design and code. Indeed, during the maintenance of systems, their structure may deteriorate because the maintainer's

efforts are focused on bugs under cost and time pressure, thus with little time to keep the code and design "clean".

Yet, code and design smells are costly because these "bad" solutions to common and recurrent problems of design and implementation are a frequent cause of low maintainability[3] [26] and can impede the evolution of the systems. During formal technical reviews, quality experts may detect errors and smells early, before they are passed on to subsequent software development and maintenance activities or released to customers. However, the detection of code and design smells requires important resources in time and personal, as well as being error-prone [31].

Code and design smells are problems resulting from bad design practices [31]. They include problems ranging from high-level and design problems, such as antipatterns [3] (that we will designate as design smells in the rest of the paper), to low-level or local problems, such as code smells [9]. Code smells are in general symptoms of design smells. However, software engineers can define their own code and design smells according to their own understanding and the context of their systems and organisations.

One example of a very common design smell is the Blob [3, p. 73–83], also known as God Class [28]. The Blob reveals a procedural design (and thinking) implemented with an object-oriented programming language. It manifests itself through a large controller class that plays a God-like role in the program by monopolizing the processing, and which is surrounded by a number of smaller data classes providing many attributes but few or no methods. In the Blob, the code smells are the large class and its surrounding data classes.

Several approaches [1,19,24] and tools [8,17,25] have been proposed in the literature to specify and detect smells. Although showing interesting results, these approaches have the following limitations :

- **Problem 1: Lack of high-level specifications of smells adapted to analysed systems.** These approaches propose the detection of a set of predefined hard-coded smells. The detection of new smells requires a new implementation using programming languages. The detection algorithms are defined at the code level by developers rather than at the domain level by quality experts. Moreover, the implementation of the detection algorithms is guided by the services of the underlying detection framework rather than by a study of the textual descriptions of the smells. Lincke *et al.* [18] demonstrate that the problem is even worst, in particular among metrics-based tools. They show that metrics-based tools not only interpret and implement the definitions of metrics differently but also deliver tool-dependent metrics results, and these differences have implications on the analyses based on the metrics results. Consequently, it is difficult for quality experts not familiar with these languages and services to evaluate the choices made by the devel-

[3] Maintainability is defined as "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment" [13].

opers, to adapt the algorithms or specify new detection algorithms to take into account the contexts of the analysed systems.

- **Problem 2: "Magical" transition from the specifications of smells to their detection.** Most of the approaches do not explain the transition from the specifications of smells to their detection. The way specifications are processed or manipulated is not clear and the definition of the underlying detection platform that allows the detection is not documented. This lack of transparency prevents replication and improvement of detection techniques, but also the technical comparison of the implementation of these techniques.

- **Problem 3: Partial validation of the existing detection techniques.** The results of the detection with the approaches are not presented on a representative set of smells and on available systems. Indeed, available results concern proprietary systems and a reduced number of smells.

In this paper, we follow the principle of domain analysis [27] to propose a domain-specific language to specify code and design smells at the domain level and generate automatically detection algorithms from these specifications. A shorter version of this paper was published in [22]. We extend this previous version of the paper with (1) a more significant set of smells, (2) detection results on a second open-source system, and (3) an extended description of the domain analysis that gives a new research point of view with respect to our previous work [21,23]. Figure 1 illustrates the following contributions:

- **Contribution 1: We describe and implement a domain-specific language, 2D-DSL** (*Design Defects Domain Specific Langage*) **based on a unified vocabulary that allows quality experts to specify smells**. This language results from a thorough domain analysis of smells, allows quality experts to take into account the contexts of the analysed systems, and allows the generation of traceable detection algorithms.

- **Contribution 2: We propose an explicit process for generating detection algorithms automatically using templates**. We recall the services provided by our underlying detection framework, 2D-FW (*Design Defects FrameWork*), which allow the automatic generation of detection algorithms from the specifications. These algorithms correspond to Java code source directly compilable and executable without any manual intervention.

- **Contribution 3: We present the validation of our process including the first study of both precision and recall on two large open-source software systems, GanttProject v1.10.2 and Xerces v2.7.0**. This validation constitutes a first result in the literature on both the precision and recall of a detection technique on open-source systems and on a representative set of 10 smells.

In the rest of this paper, Section 2 presents the domain analysis performed on the literature pertaining to smells. Section 3 presents 2D-DSL and 2D-FW.

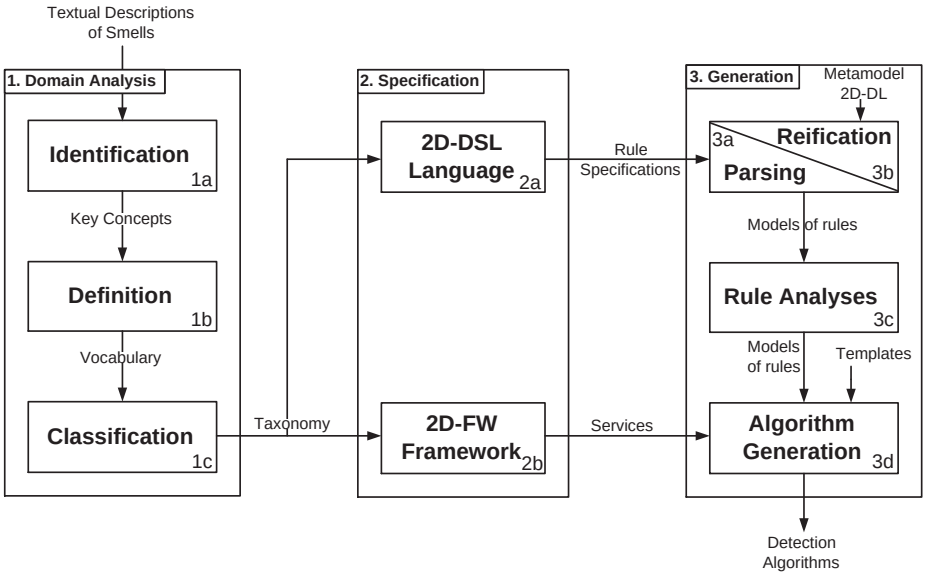

Fig. 1. Generation Process.

Section 4 describes the generation process of detection algorithms. Section 5 validates our contributions with the specification and detection of 10 smells including the antipatterns: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and the code smells: Large Class, Lazy Class, Long Method, Long Parameter List, Refused Parent Bequest, Speculative Generality, on the open-source systems Xerces v2.7.0 and GanttProject v1.10.2. Section 6 surveys related work and explicitely compares our approach to iPlasma, the current state-of-the-art tool. Section 7 concludes and presents future work.

2 Domain Analysis of Code and Design Smells

"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" [27].

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

The Blob (called also God class [28]) 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 [33]. Controller classes can be identified 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 smell 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 a ***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 smell 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 example utility classes.

Table 1. List of the Four Design Smells.

2.1 Identification of the Key Concepts

The first step of the domain analysis consists of reviewing the literature on smells for example the books and articles 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] that will be used in the rest of the paper. Table 2 gives the descriptions of the code smells participating in each of these antipatterns.

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 smell, 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 smells, which included 8 antipatterns and 21 code smells. These 29 smells are representative of the whole set of smells described in the literature and include about 60 key concepts.

The Large Class is a class with too many responsibilities. This kind of class declares a **high** number of usually unrelated methods and attributes.
The Lazy Class is a class that does not do enough. The **few** methods declared by this class have a low complexity.
The Long Method is a method with a **high** number of lines of code. A lot of variables and parameters are used. Generally, this kind of method does more than its name suggests it.
The Long Parameter List corresponds to a method with **high** number of parameters. This smell occurs when the method has more than four parameters.
The Refused Parent Bequest appears when a subclass does not use attributes and/or methods public and/or protected inherited by a parent. Typically, this means that the class hierarchy is wrong or badly organized.
The Speculative Generality is an abstract class without child classes. It was added in the system for future uses and this entity pollutes the system unnecessarily.

Table 2. List of the Six Code Smells.

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 smells. We do not present the definitions of each key concept because it would be very descriptive but we 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. In the textual descriptions of the Blob, Functional Decomposition, and Swiss Army Knife, the keywords such as *high, low, few,* and *many* are used and refer to measurable properties.

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 `CLASS_LOC`, lines of code in a method `METHOD_LOC`, 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

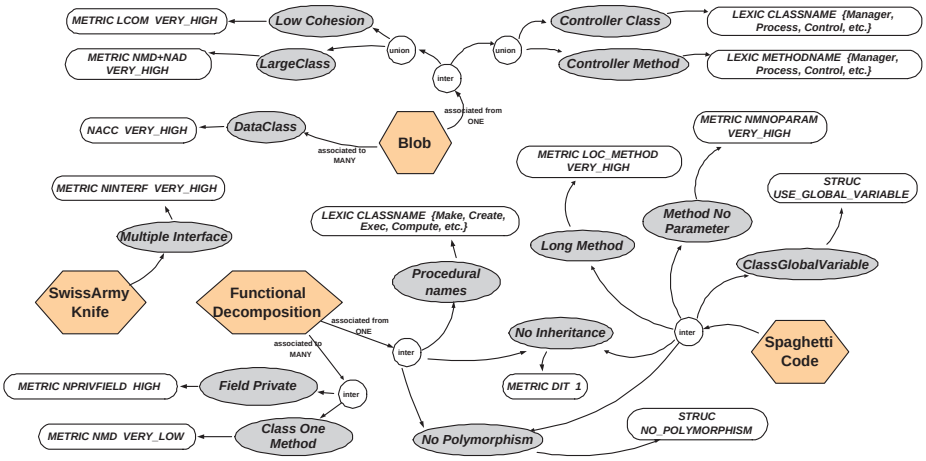

Fig. 2. Taxonomy of Smells. (Hexagons are design smells, gray ovals are code smells, white ovals are properties, and circles are set operators.)

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 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 classes 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 smells by using all relevant key concepts to relate code and design smells on a single map and clearly identify their relationships.

We produce a map organising consistently smells and key concepts. This map is important to prevent misinterpretation by clarifying and classifying smells. It is similar in purpose to Gamma *et al*'s Pattern Map [12, inside back cover]. For

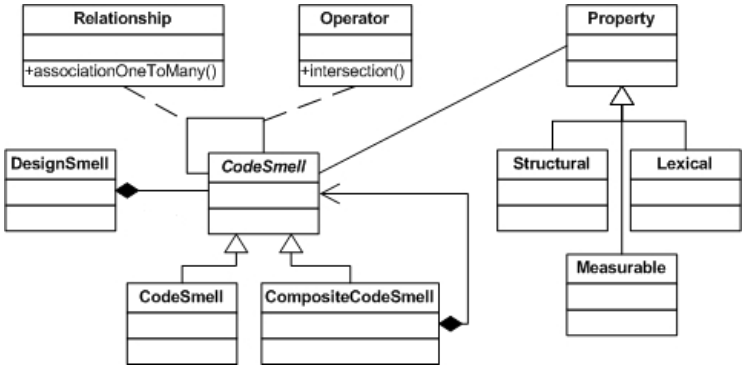

Fig. 3. The Taxonomy Meta-model.

the sake of clarity, we show in Figure 2 the taxonomy from the domain analysis of the four design smells shown in Table 1 only, which are used all through the paper. Figure 2 also includes the taxonomy of code smells associated with these design smells.

The taxonomy shows the structural relationships or set combinations (edges) among design smells (hexagons) and code smells (ovals in gray), and their relation with measurable, structural, and lexical properties (ovals in white). It gives an overview of all key concepts that characterise the four design smells 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 smells. Figure 3 gives the meta-model followed to design such a taxonomy.

3 Specification of Code and Design Smells

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 2D-DSL, a domain-specific language (DSL) [20] to specify smells in terms of their measurable, structural and lexical properties, and 2D-FW, a framework providing the services that are the operational implementations of the properties, set operations, and so on.

We build 2D-DSL and 2D-FW using and only using the identified key concepts, thus they both capture the domain expertise. Therefore, 2D-DSL and 2D-FW differ from general purpose languages and their runtime environments, which are designed to be universal [7]. They also differ from previous work on smell detection, which did not define *explicitly* a DSL and in which the detection framework drove the specification of the detection algorithms.

Thus, with 2D-DSL and 2D-FW, it is easier for quality experts to understand smell specifications and to specify new smells because these are expressed

```
1  CODESMELL    define   LongMethod        as   METRIC METHOD_LOC 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 NPRIVFIELD with    HIGH and 10.0;
9  DESIGNSMELL   define   SpaghettiCode    as  {
             ( INTER LongMethod NoParameter NoInheritance NoPolymorphism
               ProceduralName UseGlobalVariable ) };
10 DESIGNSMELL   define   FunctionalDecomposition as  {
             ASSOC FROM ( INTER ProceduralName NoInheritance NoPolymorphism) ONE
             TO ( UNION ClassOneMethod FieldPrivate) MANY };
```

Fig. 4. Specifications of the Spaghetti Code and Functional Decomposition.

using domain-related abstractions and focus on *what* to detect instead of *how* to detect it [7].

3.1 2D-DSL

With 2D-DSL, quality experts specify smells 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 design smells, which correspond to combinations of code smells using set operators or structural relationships. Figure 4 shows the specifications of the Spaghetti Code and Functional Decomposition design smells and their code smells. These two design smells shared some code smells as shown in the taxonomy in Figure 2.

A Spaghetti Code is specified using the intersection of several 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 acceptable degree of fuzziness for this measure, 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 5. A specification lists first a set of code smells and then a set of design smells (Figure 5, 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 design smells are combinations of the code smells defined in the specification using set operators (lines 5–6). The design smells can also be linked

```
1  rulespec          ::= (codesmell)+ (designsmell)+
2  codesmell         ::= CODESMELL   define codesmellName   as  csContent ;
3  designsmell       ::= DESIGNSMELL define designsmellName as  { dsContent };

4  csContent         ::= property
5  dsContent         ::= codesmellName | (operator dsContent (dsContent)+) | 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 | METHOD_LOC
11                       |CLASS_LOC | NAD | NMD | NACC | NPRIVFIELD
12                       |IR | NOC | NPrM | NOParam )
13                        ((+ | -)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 dsContent cardinality TO dsContent cardinality
22 relationshipType  ::= ASSOC | AGGREG | COMPOS
23 cardinality       ::= ONE | MANY | ONE_OR_MANY

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

Fig. 5. BNF Grammar of Smell Specifications.

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 design smells. 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 design smells.

3.2 2D-FW

The 2D-FW framework provides the services that implement operations on the relationships, operators, properties, and ordinal values. It also provides services to build, access, and analyse systems. Thus, with 2D-FW, 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. 2D-FW represents a super-set of previous detection frameworks and therefore could delegate part of its services to exiting frameworks.

2D-FW 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 [14] and accessors, and on the POM framework (*Primitives, Operators, Metrics*) for metric computation [15]. 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

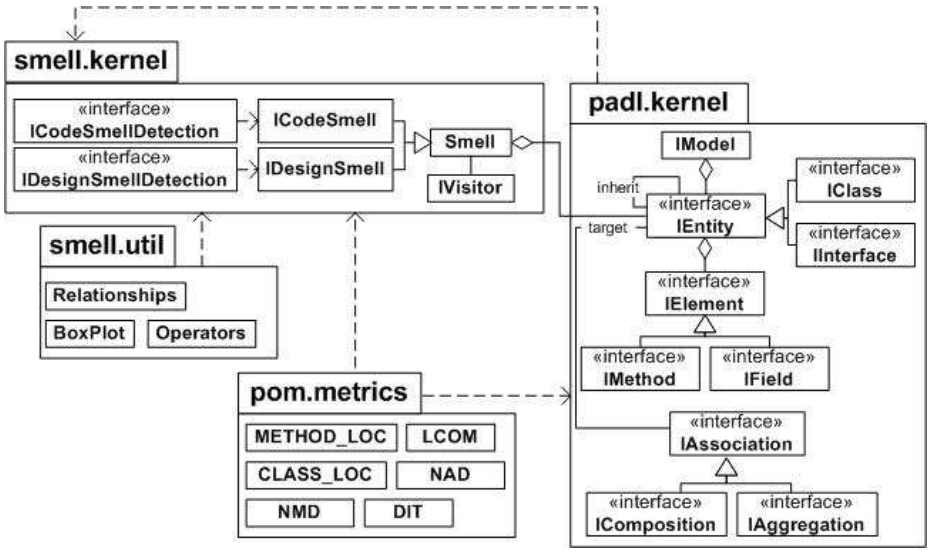

Fig. 6. Architecture of the 2D-FW Framework.

models, using the Visitor design pattern. We chose PADL because it is mature with 6 years of active development and is maintained in-house.

Figure 6 sketches the architecture of the 2D-FW framework. The architecture consists of two main packages, `smell.kernel` and `smell.util`. The `smell.kernel` package contains core classes and interfaces. Class `Smell` represents smells and is so far specialised in two subclasses, `DesignSmell` and `CodeSmell`. This hierarchy is consistent with our taxonomy of smells. A smell aggregates entities, interface `IEntity` from `padl.kernel`. For example, a smell is a set of classes with particular characteristics. Interfaces `IDesignSmellDetection` and `ICodeSmellDetection` define the services that detection algorithms must provide. The `smell.util` package declares utility classes that allow the manipulation of some key concepts of the specifications.

3.3 Discussions

The smell specifications are self-documenting and express naturally the textual descriptions of the smells. They can be modified by quality experts either to refine them by adding new rules or to modify existing ones to take into account the contexts of the systems, *i.e.*, the set of information related to the characteristics of the analysed system. The context includes the type of systems (prototypes, systems in development or maintenance, industrial systems ...), design choices (related to principles or design heuristics), and coding standards (agreements to respect during the code writing). Quality experts can thus adapt the specifications to the context of their systems because the characteristics of systems differ

from one system to the other, *e.g.*, the quantity of code comments, which can be low in prototypes but high in system in maintenance, the permitted depth of inheritance, which differs according to the design choices, and the maximal size of classes and methods defined in coding standards. Thus, specifications can be modified easily at the domain level without any knowledge of the underlying detection framework.

In the cases where the domain-specific language would not provide the required constructs to specify a certain class of smells, it is possible to extend the language with a new set of primitives. However, in this case, the underlying detection framework should also be extended to provide the operational implementations of the new primitives. Such an extension can only be realised by developers modifying the detection framework using its general-purpose implementation language.

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 detected smells. 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 smell specifications. A parser is built using JFlex and JavaCUP[4] 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 2D-DL meta-model (*Design Defects Definition Language*). The meta-model is a representation of the abstract syntax tree generated by the parser. The meta-model 2D-DL defines constituents to represent specifications, rules, set operators, relationships among rules, and properties, as illustrated in Figure 7. The result of this reification is a 2D-DL model of the specification, instance of class `Specification` defined in 2D-DL. 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 `Operator`. Structural relationships are enforced using methods defined in class `Relationship`. The 2D-DL meta-model

[4] http://www2.cs.tum.edu/projects/cup/

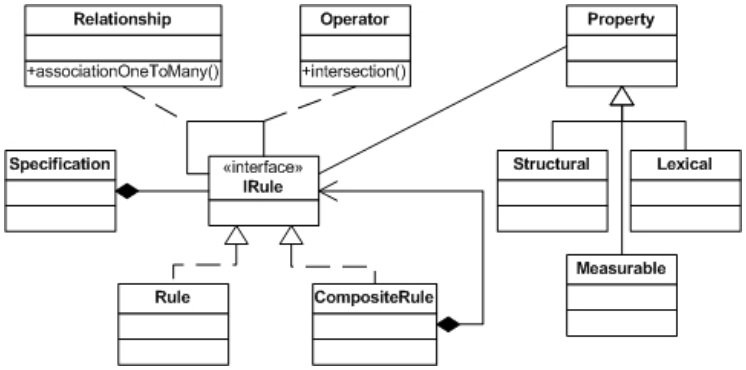

Fig. 7. The 2D-DL Meta-model.

also implements the Visitor design pattern. This meta-model is quite similar to the meta-model of the taxonomy given in Figure 3 and it illustrates the ease with which, from the taxonomy, we can specify intuitively the smells with the language.

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 with a metric has a meaning in the domain : typically, a measurable property computed with the "metric number of methods declared" `NMD` compared to a float value 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 2D-FW 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 [21,23] 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 smell and this code is directly compilable and 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 8(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 8(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, `METHOD_LOC`, tag `<FUZZINESS>` by the value 10.0, and tag `<ORDINAL_VALUES>` by the method associated with the ordinal value `VERY_HIGH`. Figure 8(g) presents the code generated for the rule given in Figure 8(a).

Set Operators. The rules can be combined using set operators such as the intersection in Figure 8(b). The code generation for set operators is quite different from the generation of properties. The template given in Figure 8(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 8(b) and returning classes that satisfy this combination. Figure 8(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 with 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 as shown in Figure 8(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 its 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 they take only few seconds.

5 Validation

We validate our contributions by specifying and detecting the 10 smells presented in Section 2 and computing the precision and recall of the generated algorithms on GanttProject v1.10.2 and Xerces v2.7.0, two open-source systems. GanttProject v1.10.2 is a project-management tool to plan projects with Gantt charts. It contains 21,267 lines of code, 188 classes, and 41 interfaces. Xerces v2.7.0 is a framework for building XML parsers in Java. It contains 71,217 lines of code, 513 classes, and 162 interfaces.

We seek in the following to obtain a recall of 100% because quality experts need all smells to improve software quality and ease formal technical reviews. Having 100% recall means that all smells are detected and, thus, that quality experts can concentrate on the classes returned by the detection without needing to analyse by hand the classes not reported because, by definition, they cannot be part of a smell.

We are not aware of any other work reporting both precision and recall for smell detection algorithms.

5.1 Validation Process

First, we build models of the analyzed systems. These models are obtained by reverse engineering. Then, we apply the generated detection algorithms on the models of the systems and obtain all suspicious classes that have potential smells. 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. The validation is inherently a manual task. Therefore, we chose to apply the detection of the six code smells on GanttProject and of the four design smells on Xerces only so that we could divide the efforts of analysing manually these systems.

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 smells, while recall assesses the number of true smells missed by the algorithms [10], according to the following equations:

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

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

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 smell or a false positive, depending on the specifications and the context and characteristics of the system. The manual analysis

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

(a) Excerpt of the Spaghetti Code.

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

(b) Excerpt of the Spaghetti Code.

```
1  public void visit(IMetric aMetric) {
2    replaceTAG("<CODESMELL>", aRule.getName());
3    replaceTAG("<METRIC>", aMetric.getName());
4    replaceTAG(<FUZZINESS>,
5      aMetric.getFuzziness());
6    replaceTAG(<ORDINAL_VALUE>,
7      aMetric.getOrdinalValue());
8  }
9  private String getOrdinalValue(int value) {
10   switch (value) {
11     case VERY_HIGH :
12       "getHighOutliers";
13     case HIGH :
14       "getHighValues";
15     case MEDIUM :
16       "getNormalValues";
17     ...
18   }
```

(c) Visitor.

```
1  public void visit(IOperator anOperator) {
2    replaceTAG("<OPERAND1>",
3      anOperator.getOperand1());
4    replaceTAG("<OPERAND2>",
5      anOperator.getOperand2());
6    switch (anOperator.getOperatorType()) {
7      case OPERATOR_UNION :
8        operator = "union";
9      case OPERATOR_INTER :
10       operator = "intersection";
11     ...
12   }
13   replaceTAG("<OPERATION>", operator);
```

(d) Visitor.

```
1  public class <CODESMELL>Detection
2    extends CodeSmellDetection
3    implements ICodeSmellDetection {
4    public Set performDetection() {
5      IClass c = iteratorOnClasses.next();
6      LOCofSetOfClasses.add(
7        Metrics.compute(<METRIC>, c));
8      ...
9      BoxPlot boxPlot = new BoxPlot(
10       <METRIC>ofSetOfClasses, <FUZZINESS>);
11     Map setOfOutliers =
12       boxPlot.<ORDINAL_VALUE>();
13     ...
14     suspiciousCodeSmells.add( new CodeSmell(
15       <CODESMELL>, setOfOutliers));
16     ...
17     return suspiciousCodeSmells;
18   }
```

(e) Template.

```
1  public class <CODESMELL>Detection
2    extends CodeSmellDetection
3    implements ICodeSmellDetection {
4    public void performDetection() {
5      ICodeSmellDetection cs<OPERAND1> =
6        new <OPERAND1>Detection();
7      op1.performDetection();
8      Set set<OPERAND1> =
9        cs<OPERAND1>.listOfCodeSmells();
10     ICodeSmellDetection cs<OPERAND2> =
11       new <OPERAND2>Detection();
12     op2.performDetection();
13     Set set<OPERAND2> =
14       cs<OPERAND2>.listOfCodeSmells();
15     Set setOperation = Operators.getInstance().
16       <OPERATION>(set<OPERAND1>, set<OPERAND2>);
17     this.setSetOfSmells(setOperation);
18   }
```

(f) Template.

```
1  public class LongMethodDetection
2    extends CodeSmellDetection
3    implements ICodeSmellDetection {
4    public Set performDetection() {
5      IClass c = iteratorOnClasses.next();
6      LOCofSetOfClasses.add(
7      Metrics.compute("LOC_METHOD", c));
8      ...
9      BoxPlot boxPlot = new BoxPlot(
10       LOC_METHODofSetOfClasses, 10.0);
11     Map setOfOutliers =
12       boxPlot.getHighOutliers();
13     ...
14     suspiciousCodeSmells.add( new CodeSmell(
15       LongMethod, setOfOutliers));
16     ...
17     return suspiciousCodeSmells;
18   }
```

(g) Generated Code.

```
1  public class Inter1
2    extends CodeSmellDetection
3    implements ICodeSmellDetection {
4    public void performDetection() {
5      ICodeSmellDetection csLongMethod =
6        new LongMethodDetection();
7      csLongMethod.performDetection();
8      Set setLongMethod =
9        csLongMethod.listOfCodeSmells();
10     ICodeSmellDetection csNoParameter =
11       new NoParameterDetection();
12     csNoParameter.performDetection();
13     Set setNoParameter =
14       csNoParameter.listOfCodeSmells();
15     Set setOperation = Operators.getInstance().
16       intersection(setLongMethod, setNoParameter);
17     this.setSetOfSmells(setOperation);
18   }
```

(h) Generated Code.

Fig. 8. Code Generation for Measurable Properties (left) and Set Operators (right).

Code Smells	Numbers of Known True Positives		Numbers of Detected Smells		Precision	Recall	Time
Large Class	9	(4.79%)	13	(6.91%)	69.23%	100.00%	0.008s
Lazy Class	41	(21.81%)	104	(55.32%)	34.61%	87.80%	0.3s
Long Method	45	(23.94%)	22	(11.70%)	46.66%	95.45%	0.08s
Long Parameter List	54	(28.72%)	43	(22.87%)	79.63%	100.00%	0.030s
Refused Parent Bequest	18	(9.57%)	20	(10.64%)	40.00%	44.45%	0.16s
Speculative Generality	4	(2.13%)	4	(2.13%)	100.00%	100.00%	0.01s

Table 3. Precision and Recall of Code Smells in GanttProject v1.10.2. (In parenthesis, the percentage of classes affected by a smell). The number of classes in GanttProject v1.10.2 is 188.

Design Smells	Numbers of Known True Positives		Numbers of Detected Smells		Precision	Recall	Time
Blob	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 4. Precision and Recall of Design Smells in Xerces v2.7.0. (In parenthesis, the percentage of classes affected by a smell.). The number of classes in Xerces v2.7.0 is 513.

of GanttProject has been performed by one independent software engineer with a good expertise in code and design smells. We asked three master students and two independent software engineers to analyse manually Xerces using only Brown and Fowler's books to identify smells and compute the precision and recall of the algorithms. Each time a doubt on a candidate class arose, the software engineers considered the books as references in deciding by consensus whether or not this class was actually a smell. This task is tedious, some smells 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 4 and 3 report detection times, numbers of suspicious classes, and precisions and recalls, respectively for the GanttProject and Xerces systems. 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.

Excluding the code smells Lazy Class and Refused Parent Bequest, the recalls of the generated algorithms range from 95.45% to 100% for code smells and 100% for design smells. Precisions range from 46.66% to 100%, providing between 2.13% and 22.87% of the total number of classes, which is reasonable for quality experts to analyse by hand, with respect to analysing the entire system, 188 classes GanttProject, 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 4). The result file contains all suspicious classes, including class org.apache.xerces.xinclude.-XIncludeHandler declaring 112 methods. Among these 112 methods, method `handleIncludeElement(XMLAttributes)` is typical of Spaghetti Code: it does not use inheritance and polymorphism, uses excessively global variables, and weighs 759 LOC.

For the Large Class, Long Parameter List, and Speculative Generality code smells, we obtained optimum recalls because the generated detection algorithms are based on quite simple metrics that consist in counting entities. For the Long Method, the same kind of simple metric, the number of lines of code, has been used. However, this metric depends on the experts' definition of a *long* method, which explains that we did not obtain a perfect recall (95.45%).

We encountered another ambiguity with Lazy Class. Indeed, one of the criteria of its detection is method complexity: during manual detection, software engineers must appreciate method complexity, which is inherently subjective, while we use objective metrics (WMC, Weighted methods per class, and McCabe Cyclomatic Complexity) during the automatic detection.

Finally, the smell Refused Parent Bequest illustrated the inverse problem: it is very difficult for software engineers to identify all its occurrences because they must appreciate if a class uses *enough* public and protected methods/fields of any of its superclasses. Moreover, the software engineers consider bequests provided by API classes whereas we apply our detection on the system only, excluding libraries.

The two smells Lazy Class and Refused Parent Bequest exhibit poorer precision and recall because of the lack of constraints of their specifications. A Lazy Class could be any class with *few simple methods* while a Refused Parent Bequest could be any class that does not directly use some of its parents methods. These two smells illustrate the problems of interpreting the definitions of the smells and of translating these definitions into specifications. Future work includes studying the contexts in which these specifications could be made more constraining.

The results depend on the specifications of the smells. The specifications must be neither too loose, not to detect too many suspicious classes, nor too constraining, and miss smells. With 2D-DSL, quality experts can refine the specifications of the smells easily, according to the detected suspicious classes and their knowledge of the system through iterative refinement.

5.3 Discussions

Results. The validation shows that the specifications of the design and code smells lead to generated detection algorithms with expected recalls and good precisions. Thus, it confirms that: (1) the language allows describing several smells. We described 10 smells including 4 different design smells and 6 code smells; (2) the generated detection algorithms have, *mainly*, a recall of 100%, *i.e.*, all known smells are detected, and an average precision greater than 34%,

i.e., the detection algorithms report less than 2/3 of false positives with respect to the number of true positives. Quality experts do not need to look at classes not detected as smells because they cannot be part of smells thanks to the recall of 100%; 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.

Threats to the Validity. The validity of the results depends directly on the smell specifications. We performed our experiments on a representative set of smells to lessen the threat to the internal validity of our validation. The subjective nature of interpreting, specifying, and identifying smells is a threat to the construct validity of this validation. We lessen this threat by specifying the smells based on the literature and by involving several independent software engineers in the computation and manual assessment of the results. The threat to external validity is related to the exclusive use of open-source JAVA systems, which may prevent the generalisation of our results to other systems. Nevertheless, we chose to perform our experiments on freely available systems to allow their verification and replication. Furthermore, in the context of a cooperation with a company (not described here for confidentiality reasons), we also applied our approach on their proprietary systems and we obtained similar encouraging results.

6 Related Work and Current State-of-the-Art

Several smell detection approaches and tools have been proposed in the literature to specify and detect code and design smells. However, we are not aware of any approach that is based on an explicit domain analysis and its resulting domain-specific language. Moreover, most of these approaches do not clarify the process that allows the detection of smells from their specifications. These approaches are often driven by the services of their underlying detection framework rather than by an exhaustive study of the descriptions of smells that led to their specifications. Finally, if others explain their detection process, they do not provide a validation of their detection technique. We first present the related work and then a detailed comparison of our approach with the approach of detection strategies [19], the current state-of-the-art.

6.1 Related Work

Several books provide in-breadth views on pitfalls [32], heuristics [28], code smells [9], and antipatterns [3] aimed at a wide audience for educational purposes. However, they describe *textually* smells but none performed a complete analysis of the text-based descriptions of smells. Thus, it is difficult to build detection

algorithms from their textual descriptions because they lack precision and are prone to misinterpretation. Travassos *et al.* [31] introduced a process based on manual inspections and reading techniques to identify smells. 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 smells, not their specification.

Marinescu introduced the concept of detection strategies to overcome the limitations of the previous literature. We present a detailed comparison of our approach with these detection strategies in the following subsection. Munro [24] also noticed the limitations of the textual descriptions and proposed a template including heuristics 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. Only Marinescu [19] and Munro [24] provide some results of their detection but on a few smells and only on proprietary systems. Alikacem and Sahraoui proposed a description language of quality rules to detect violations of quality principles and smells [1]. The rules include quantitative information such as metrics and structural information such as inheritance or association relationships among classes. They also use fuzzy logic to express the thresholds of rules conditions. The rules are interpreted and executed using an inference engine. Although their detection framework is explicit, they did not provide any validation of their approach. All these approaches have contributed significantly to the automatic detection of smells. However, none presents a complete solution including a specification language, explicit detection framework detailed processing, and validation of the detection technique.

Tools such as PMD [25], 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 smells such as antipatterns. PMD [25], SemmleCode [29] and Hammurapi [16] also allow developers to write detection rules using Java or XPath. However, the addition of new rules is intended for engineers familiar with Java and XPath, which could limit access to a wider range of software engineers. 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. Therefore, previous tools provided low-level languages for specifying new smells while we defined a high-level DSL targeting software quality experts.

6.2 Comparison with iPlasma

Marinescu [19] presented a metric-based approach to detect code smells with the concept of *detection strategies* and implemented a tool called iPlasma. Detection strategies capture deviations from good design principles. They are described with a medium-level specification language based on metrics through a simple process. They are therefore a step towards the precise specifications

Code Smells	Numbers of Known True Positives		Numbers of Detected Smells		Precision	Recall
Large Class	9	(4.79%)	1	(0.53%)	100.00%	11.11%
Lazy Class	41	(21.81%)	∅		∅	∅
Long Method	45	(23.94%)	11	(5.85%)	100.00%	24.44%
Long Parameter List	54	(28.72%)	44	(23.40%)	100.00%	81.48%
Refused Parent Bequest	18	(9.57%)	1	(0.53%)	0%	0%
Speculative Generality	4	(2.13%)	9	(4.79%)	100.00%	44.44%

Table 5. Precision and Recall of Code Smells in GanttProject v1.10.2 using iPlasma.

of code smells and, as such, have been often cited. Therefore, we considered as others detection strategies and iPlasma the current state-of-the-art. Yet, detection strategies have limitations. First, the specifications are mainly metric-based and, thus for example, cannot describe specific relationships between classes of interest. Second, the specifications are hard-coded in the tool and are not readily available for study and–or modification.

To further compare the two approaches, we have proceeded to the detection of the six code smells presented in Table 2 using iPlasma on GanttProject as for our approach in Table 3. Table 5 presents the results of this detection.

First, we notice that iPlasma also uses the technique of the boxplot to evaluate relative values specified in the detection strategies. However, the mapping from the relative values of the boxplot with the metric values is not explicit. Therefore, we could not detect occurrences of Lazy Class because it was not possible to define explicitly rules using the bloxplot. In our approach, we make explicit the boxplot and enhance it with fuzzy logic and, thus, alleviates the problem related to the definition of thresholds.

Second, iPlasma focuses on a high precision rather that a high recall. This is an appropriate choice in the context of a day-to-day use by developers as part of their development process. In our approach, we focus on a high recall but developers could adapt the detection algorithms to enforce a high precision using our high-level domain-specific language.

Third, the results of the detection of occurrences of Refused Bequest Parent in terms of precision and recall are surprising. iPlasma only detects one (false-positive) occurrence while, our approach provides a 40% precision and 44% recall. We cannot explain this poor performance of iPlasma for lack of access to the specifications of the smells. Other code smells exhibit perfect precisions and recalls between 11% and 81%.

Finally, although the computation times in iPlasma are similar to those of our approach (typically less than a second), iPlasma requires user interactions to select the smells and the programs in which to detect their occurrences while our approach can be fully automated and integrated in the developers' development environment.

In conclusion, we drew much inspiration from the concept of detection strategies and from their implementation in iPlasma to design a high-level domain specific language and its companion generation algorithms. We thus attempted to overcome the limitations of this approach illustrated in this comparison.

7 Conclusion and Future Work

In this paper, we introduced a domain analysis of the key concepts defining design and code smells; a domain-specific language to specify smells, 2D-DSL, and its underlying detection framework, 2D-FW; and a process for generating detection algorithms automatically.

With 2D-DSL, 2D-FW, and the generation process, quality experts can specify smells at the domain level using their expertise, generate detection algorithms, and detect smells during formal technical reviews. We implemented 2D-DSL, 2D-FW, and the generation process and studied the precision and recall of 10 generated algorithms on GanttProject v1.10.2 and Xerces v2.7.0. We showed that the detection algorithms are efficient and precise, and have a recall close to 100% recall.

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 in complement to the comparison with iPlasma described here. We also plan to integrate the WordNet lexical database of English into 2D-FW, to improve the current DSL, 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 [30]. 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 2D-FW.

References

1. El H. Alikacem and H. Sahraoui. Détection d'anomalies utilisant un langage de description de règle de qualité. In *actes du* 12^e *colloque LMO*, pages 185–200, 2006.
2. D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *Transactions on Software Engineering*, 31(2):137–149, 2005.
3. W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, March 1998.
4. J. M. Chambers, W. S. Clevelmd, B. Kleiner, and P. A. Tukey. *Graphical methods for data analysis*. Wadsworth International, 1983.
5. CheckStyle, 2004. http://checkstyle.sourceforge.net.
6. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
7. C. Consel and R. Marlet. Architecturing software using: A methodology for language development. *LNCS*, 1490:170–194, 1998.
8. G. Florijn. Revjava – design critiques and architectural conformance checking for java software, 2002. White Paper. SERC, the Netherlands.

9. M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1^{st} edition, 1999.
10. W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.
11. FXCop, 2006. http://www.gotdotnet.com/team/fxcop/.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1^{st} edition, 1994.
13. A. Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers, Inc., 1991.
14. Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Proceedings of the* 19^{th} *OOSPLA Conference*, pages 301–314, 2004.
15. Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *Proceedings of the* 11^{th} *WCRE Conference*, pages 172–181, 2004.
16. Hammurapi, October 2007. http://www.hammurapi.biz/.
17. D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
18. R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the ISSTA Symposium*, pages 131–142, New York, NY, USA, 2008. ACM.
19. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *ICSM Conference*, pages 350–359, 2004.
20. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
21. N. Moha, Y.-G. Guéhéneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Proceedings of the* 21^{st} *ASE Conference*, 2006.
22. N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proceedings of the* 11^{th} *FASE Conference*, pages 276–291. Springer-Verlag, 2008.
23. 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 LMO*, pages 201–216, 2006.
24. M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proceedings of the* 11^{th} *Metrics Symposium*, 2005.
25. PMD, 2002. http://pmd.sourceforge.net/.
26. R. S. Pressman. *Software Engineering – A Practitioner's Approach.* McGraw-Hill Higher Education, 5^{th} edition, November 2001.
27. R. Prieto-Díaz. Domain analysis: An introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
28. A. J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.
29. SemmleCode, October 2007. http://semmle.com/.
30. G. S. Swint, C. Pu, G. Jung, W. Yan, Y. Koh, Q. Wu, C. Consel, A. Sahai, and K. Moriyama. Clearwater: extensible, flexible, modular code generation. In *Proceedings of the 20th IEEE/ACM ASE Conference*, pages 144–153, 2005.
31. G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the* 14^{th} *OOSPLA Conference*, pages 47–56, 1999.
32. B. F. Webster. *Pitfalls of Object Oriented Development.* M & T Books, 1995.
33. R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities and Collaborations.* Addison-Wesley Professional, 2002.