DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

Abstract— **Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: (1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells; (2) DETEX, a detection technique that instantiates this method; and (3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high-level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Using DETEX, we specify four well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on XERCES v2.7.0, and discuss the precision of these algorithms on 11 open-source systems.**

Index Terms— **Antipatterns, design smells, code smells, specification, meta-modelling, detection, JAVA.**

I. INTRODUCTION

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, opposite to design patterns [1], code and design smells—“poor” solutions to recurring implementation and design problems—may hinder their evolution by making it hard for software engineers to carry out changes.

Code and design smells include low-level or local problems such as code smells [2], which are usually symptoms of more global design smells such as antipatterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [2] presented 22 code smells, structures in the source code that suggest the possibility of refactorings. Duplicated code, long methods, large classes, and long parameter lists, are just a few symptoms of design smells and opportunities for refactorings.

One example of a design smell is the Spaghetti Code antipattern[1], which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-orientation mechanisms such as polymorphism and inheritance and prevents their use.

We use the term “smells” to denote both code and design smells. This use does not exclude that, in a particular context, a smell can be the best way to actually design or implement a system. For example, parsers generated automatically by parser generators are often Spaghetti Code, *i.e.*, very large classes with very long methods. Yet, although such classes “smell”, software engineers must manually evaluate their possible negative impact according to the context.

The detection of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [4]. However, detection in large systems is a very time- and resource-consuming and error-prone activity [5] because smells cut across classes and methods and their descriptions leave much room for interpretation.

Several approaches, as detailed in Section II, have been proposed to specify and detect smells. However, they have three limitations. First, the authors do not explain the analysis leading to the specifications of smells and the underlying detection framework. Second, the translation of the specifications into detection algorithms is often black-box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of smells and systems to allow comparison among approaches. So far, reported results concern proprietary systems and a reduced number of smells.

We present three contributions to overcome these limitations. First, we propose DECOR (*DEtection & CORrection*[2]), a method that describes all the steps necessary for the specification and detection of code and design smells. This method embodies in a coherent whole all the steps defined by previous work and, thus, provides a means to compare existing techniques and to suggest future work.

Second, we revisit in the context of the DECOR method our detection technique [6], [7], now called DETEX (*DETection EXpert*). DETEX allows software engineers to specify smells at a high-level of abstraction using a unified vocabulary and domain-specific language, obtained from an in-depth domain analysis, and to automatically generate detection algorithms. Thus, DECOR represents a concrete and generic method for the detection of smells with respect to previous work and DETEX is an instantiation or a concrete implementation of this method in the form of a detection technique.

N. Moha, post-doctoral fellow in INRIA – Triskell Team, IRISA - Université de Rennes 1, France. E-mail: moha@irisa.fr

Y.-G. Guéhéneuc, Associate Professor, in Département de génie informatique et génie logiciel, École Polytechnique de Montréal, Québec, Canada. E-mail: yann-gael.gueheneuc@polymtl.ca

L. Duchien, Professor, and A.-F. Le Meur, Associate Professor, in LIFL, INRIA Lille-Nord Europe – ADAM Team, Université de Lille, France. E-mail:{Laurence.Duchien, Anne-Francoise.Le-Meur}@lifl.fr

[1]This smell, like those presented later on, is really in between implementation and design.

[2]Correction is future work.

Third, we validate DETEX using precision and recall on the open-source system XERCES and precision on 11 other systems. We thus show indirectly the usefulness of DECOR. This extensive validation is the first report in the literature of both precision and recall with open-source software systems.

These three contributions take up and expand our previous work on code and design smells [6], [7] to form a consistent whole that provides all the necessary details to understand, use, replicate, and pursue our work. Therefore, we take up the domain analysis, language, underlying detection framework, and results of the recall on XERCES.

The paper is organized as follows. Section II surveys related work. Section III describes the DECOR method and introduces its instantiation DETEX. Section IV details each step of the implementation of DETEX illustrated on the Spaghetti Code as a running example. Section V describes the validation of DETEX with the specification and detection of three additional design smells: Blob, Functional Decomposition, and Swiss Army Knife, on 11 object-oriented systems: ARGOUML, AZUREUS, GANTTPROJECT, LOG4J, LUCENE, NUTCH, PMD, QUICKUML, two versions of XERCES, and ECLIPSE. Section VI concludes and presents future work.

II. RELATED WORK

Many works exist on the identification of problems in software testing [8], databases ([9], [10]), and networks [11]. We survey here those works directly related to the detection of smells by presenting their existing descriptions, detection techniques, and related tools. Related work on design pattern identification (*e.g.*, [12]) is beyond the scope of this paper.

A. Descriptions of Smells

Several books have been written on smells. Webster [13] wrote the first book on smells in the context of object-oriented programming, including conceptual, political, coding, and quality-assurance pitfalls. Riel [14] defined 61 heuristics characterising good object-oriented programming that enable engineers to assess the quality of their systems manually and provide a basis for improving design and implementation. Beck in Fowler's book [2] compiled 22 code smells that are low-level design problems in source code, suggesting that engineers should apply refactorings. Code smells are described in an informal style and associated with a manual detection process. Mäntylä [15] and Wake [16] proposed classifications for code smells. Brown *et al.* [3] focused on the design and implementation of object-oriented systems and described 40 antipatterns textually, *i.e.*, general design smells including the well-known Blob and Spaghetti Code.

These books provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide academic audience. However, manual inspection of the code for searching for smells based only on text-based descriptions is a time-consuming and error-prone activity. Thus, some researchers have proposed smell detection approaches.

B. Detection Techniques

Travassos *et al.* [5] 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, the process only covers the manual detection of smells, not their specification.

Marinescu [17] presented a metric-based approach to detect code smells with detection strategies, implemented in the IPLASMA tool. The strategies capture deviations from good design principles and consist of combining metrics with set operators and comparing their values against absolute and relative thresholds.

Munro [18] noticed the limitations of text-based descriptions and proposed a template to describe code smells systematically. This template is similar to the one used for design patterns [1]. It consists of three main parts: a code smell name, a text-based description of its characteristics, and heuristics for its detection. It is a step towards more precise specifications of code smells. Munro also proposed metric-based heuristics to detect code smells, which are similar to Marinescu's detection strategies. He also performed an empirical study to justify the choice of metrics and thresholds for detecting smells.

Alikacem and Sahraoui [19] proposed a language to detect violations of quality principles and smells in object-oriented systems. This language allows the specification of rules using metrics, inheritance or association relationships among classes, according to the engineers' expectations. It also allows using fuzzy logic to express the thresholds of rules conditions. The rules are executed by an inference engine.

Some approaches for complex software analysis use visualisation techniques [20], [21]. Such semi-automatic approaches are interesting compromises between fully automatic detection techniques that can be efficient but loose track of context, and manual inspection that is slow and inaccurate [22]. However, they require human expertise and are thus still time-consuming. Other approaches perform fully automatic detection of smells and use visualisation techniques to present the detection results [23], [24].

Other related approaches include architectural consistency checkers, which have been integrated in style-oriented architectural development environments [25], [26], [27]. For example, active agents acting as critics [27] can check properties of architectural descriptions, identify potential syntactic and semantic errors, and report them to the designer.

All these approaches have contributed significantly to the automatic detection of smells. However, none presents a complete method including a specification language, an explicit detection platform, a detailed processing, and a validation of the detection technique.

C. Tools

In addition to detection techniques, several tools have been developed to find smells and implementation problems and–or syntax errors.

Annotation checkers such as ASPECT [28], LCLINT [29], or EXTENDED STATIC CHECKER [30] use program verification techniques to identify code smells. These tools require the engineers' assistance to add annotations in the code that can be used to verify the correctness of the system.

SMALLLINT [31] analyses Smalltalk code to detect bugs, possible programming errors, or unused code. FINDBUGS [32] detects correctness- and performance-related bugs in JAVA systems. SABER [33] detects latent coding errors in J2EE-based applications. ANALYST4J [34] allows the identification of antipatterns and code smells in JAVA systems using metrics. PMD [35], CHECKSTYLE [36], and FXCOP [37] check coding styles. PMD [35] and HAMMURAPI [38] 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 audience. With SEMMLECODE [39], engineers can execute queries against source code, using a declarative query language called .QL, to detect code smells.

CROCOPAT [40] provides means to manipulate relations of any arity with a simple and expressive query and manipulation language. This tool allows many structural analyses in models of object-oriented systems including design patterns identification and detection of problems in code (for example, cycles, clones, and dead code).

Model checkers such as BLAST [41] and MOPS [42] also relate to code problems by checking for violations of temporal safety properties in C systems using model checking techniques.

Most of these tools detect predefined smells at the implementation level such as bugs or coding errors. Some of them as PMD [35] and HAMMURAPI [38] allow engineers to specify new detection rules for smells using languages such as JAVA or XPATH.

III. DECOR AND ITS INSTANTIATION, DETEX

Although previous works offer ways to specify and detect code and design smells, each has its particular advantages and focuses on a subset of all the steps necessary to define a detection technique systematically. The processes used and choices made to specify and implement the smell detection algorithms are often not explicit: they are often driven by the services of the underlying detection framework rather than by an exhaustive study of the smell descriptions.

Therefore, as a first contribution, we propose DECOR, a method that subsumes all the steps necessary to define a detection technique. The method defines explicitly each step to build a detection technique. All steps of DECOR are partially instantiated by the previous approaches. Thus, the method encompasses previous work in a coherent whole. Figure 1 (a) shows the five steps of the method. The following items summarise its steps:

- **Step 1. Description Analysis**: Key concepts are identified in the text-based descriptions of smells in the literature. They form a unified vocabulary of reusable concepts to describe smells.
- **Step 2. Specification**: The concepts, which constitute a vocabulary, are combined to specify smells systematically and consistently.
- **Step 3. Processing**: The specifications are translated into algorithms that can be directly applied for the detection.
- **Step 4. Detection**: The detection is performed on systems using the specifications previously processed and returns the list of code constituents (*e.g.*, classes, methods) suspected of having smells.
- **Step 5. Validation**: The suspected code constituents are manually validated to verify that they have real smells.

The first step of the method is generic and must be based on a representative set of smells. Steps 2 and 3 must be followed when specifying a new smell. The last two steps 4 and 5 are repeatable and must be applied on each system. Feedback loops exist among the steps when the validation of the output of a step suggests changing the output of its precursor.

During the iterative validation, we proceed as follows: In Step 1, we may expand the vocabulary of smells; In Step 2, we may extend the specification language; In Step 3, we may refine and reprocess the specifications to reduce the number of erroneous detection results. The engineers choose the stopping criteria depending on their needs and the outputs of the detection. Steps 1, 2 and 5 remain manual by nature.

Figure 1 (a) contrasts the DECOR method with previous work. Some previous work [2], [3], [13], [14] provided text-based descriptions of smells but none performed a complete analysis of these descriptions. Munro [18] improved the descriptions by proposing a template including heuristics for their detection. However, he did not propose any automatic process for their detection. Marinescu [17] proposed a detection technique based on high-level specifications. However, he did not make explicit the processing of these specifications, which appears as a black box. Alikacem and Sahraoui [19] also proposed high-level specifications but did not provide any validation of their approach. Tools focused on implementation problems and could provide hints on smells and thus implement parts of the detection. Although these tools provide languages for specifying new smells, these specifications are intended for developers and, thus, are not high-level specifications. Only Marinescu [17] and Munro [18] provide some results of their detection but only on a few smells and proprietary systems.

As our second contribution, we now revisit our previous detection technique [6], [7] within the context of DECOR. Figure 1 (b) presents an overview of the four steps of DETEX, which are instances of the steps of DECOR. It also emphasises the steps, inputs, and outputs specific to DETEX. The following items summarise the steps in DETEX:

- **Step 1. Domain Analysis**: This first step consists of performing a thorough analysis of the domain related to smells to identify key concepts in their text-based descriptions. In addition to a unified vocabulary of reusable concepts, a taxonomy and classification of smells are defined using the key concepts. The taxonomy highlights and charts the similarities and differences among smells and their key concepts.
- **Step 2. Specification**: The specification is performed using a domain-specific language (DSL) in the form of *rule cards* using the previous vocabulary and taxonomy. A rule card is a set of rules. A rule describes the properties that a class must have to be considered a smell. The DSL allows defining properties for the detection of smells, specifying the structural relationships among these properties and characterising properties according to their lexicon (*i.e.*, names), structure (*e.g.*, classes using global variables), and internal attributes using metrics.

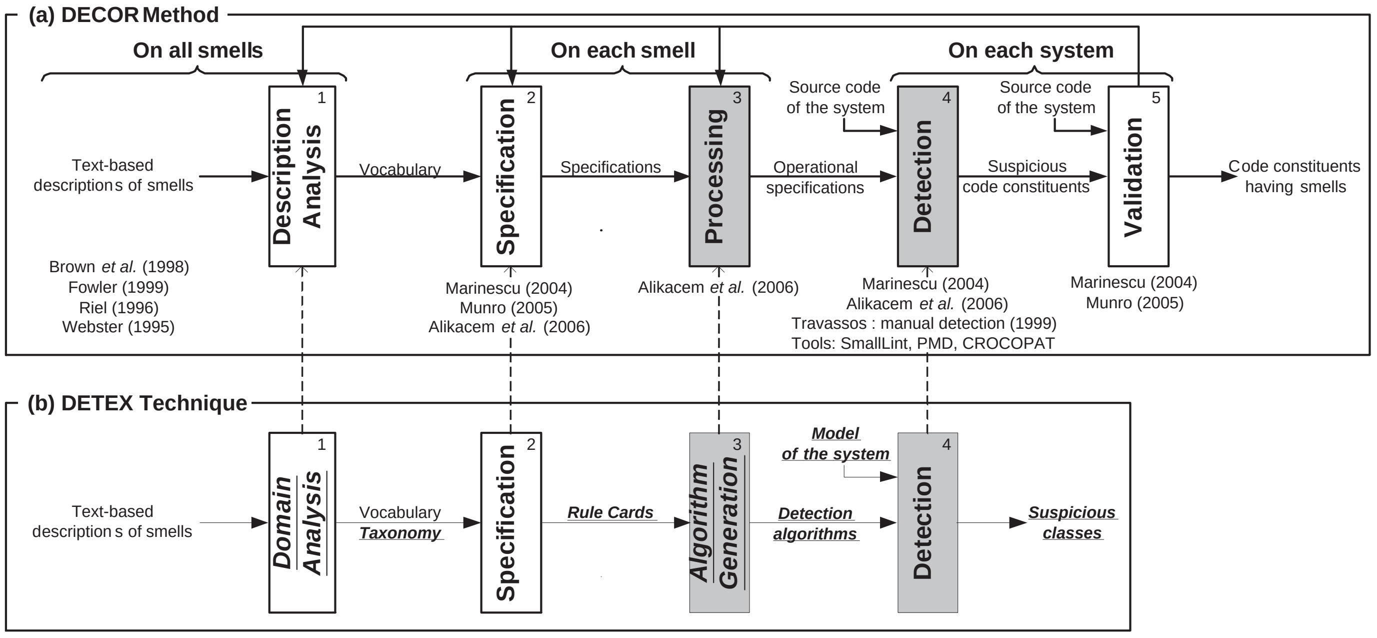

Fig. 1. (a) DECOR Method Compared to Related Work. (Boxes are steps and arrows connect the inputs and outputs of each step. Gray boxes represent fully-automated steps.)
(b) DETEX Detection Technique. (The steps, inputs, and outputs in bold, italics, and underlined are specific to DETEX compared with DECOR.)

- **Step 3. Algorithm Generation**: Detection algorithms are automatically generated from models of the rule cards. These models are obtained by reifying the rules using a dedicated meta-model and parser. A framework supports the automatic generation of the detection algorithms.
- **Step 4. Detection**: Detection algorithms are applied automatically on models of systems obtained from original designs produced during forward engineering or through reverse engineering of the source code.

DETEX is original because the detection algorithms are not ad hoc but generated using a DSL obtained from an in-depth domain analysis of smells. A DSL benefits the domain experts, engineers, and quality experts because they can specify and modify manually the detection rules using high-level abstractions pertaining to their domain, taking into account the context of the analysed systems. The context corresponds to all information related to the characteristics of the systems including types (prototype, system in development or maintenance, embedded system, etc.), design choices (related to design heuristics and principles), and coding standards.

IV. DETEX IN DETAILS

The following sections describe the four steps of DETEX using a common pattern: input, output, description, and implementation. Each step is illustrated by a running example using the Spaghetti Code and followed by a discussion.

A. Step 1: Domain Analysis

The first step of DETEX is inspired by the activities suggested for domain analysis [43], which "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". In the context of smells, *information* relates to smells, *software systems* are detection algorithms, and the information on smells must be *reusable* when specifying new smells. Domain analysis ensures that the language for specifying smells is built upon consistent high-level abstractions and is flexible and expressive. This step is crucial to DETEX because its output serves as input for all the following steps. In particular, the identified key concepts will be specified as properties and values in the next two steps.

1) Process:

Input: Text-based descriptions of design and code smells in the literature, such as [2], [3], [13], [14].

Output: A textual list of the key concepts used in the literature to describe smells, which forms a vocabulary for smells. Also, a classification of code and design smells and a taxonomy in the form of a map highlighting similarities, differences, and relationships among smells.

Description: This first step deals with identifying, defining, and organising key concepts used to describe smells, including metric-based heuristics as well as structural and lexical data [7]. The key concepts refer to keywords or specific concepts of object-oriented programming used to describe smells in the literature ([2], [3], [13], [14]). They form a vocabulary of reusable concepts to specify smells.

The domain analysis requires a thorough search of the literature for key concepts in the smell descriptions. We perform the analysis in an iterative way: for each description of a smell, we extract all key concepts, compare them with already-found concepts, and add them to the domain avoiding synonyms and homonyms. A synonym is a same concept with two different names and homonyms are two different concepts with a same name. Thus, we obtain a compilation of concepts that form a concise and unified vocabulary.

We define and classify manually smells using the key concepts. Smells sharing the same concepts belong to the same category. The classification limits possible misinterpretation, avoiding synonyms and homonyms at any level of granularity. We sort the concepts according to the types of properties on which they apply: measurable, lexical, or structural.

Measurable properties are concepts expressed with measures of internal attributes of constituents of systems (classes, methods, fields, relationships, and so on). Lexical properties relate to the vocabulary used to name constituents. They characterise constituents with specific names defined in lists of keywords or in a thesaurus. Structural properties and relationships define the structures of constituents (for example, fields corresponding to global variables) and their relationships (for example, an association relationship between classes).

Figures 2 and 3 show the classifications of the four antipatterns of interest in this paper, described in Table I, and their code smells. These classifications organise and structure smells consistently at the different levels of granularity.

We then use the vocabulary to manually organise all smells with respect to one another and build a taxonomy that puts all smells on a single map and highlights their relationships. The map organises and combines smells, such as antipatterns and code smells, and other related key concepts using set operators such as intersection and union.

Implementation: This step is intrinsically manual. It requires the engineers' expertise and can seldom be supported by tools.

2) Running Example:

Analysis of the Spaghetti Code: We summarise the text-based description of the Spaghetti Code [3, page 119] in Table I along with those of the Blob [page 73], Functional Decomposition [page 97], and Swiss Army Knife [page 197]. In the description of the Spaghetti Code, we identify the key concepts (in italic in the table) of classes with long methods with no parameter, with procedural names, declaring global variables, and not using inheritance and polymorphism.

We obtain the following classification for the Spaghetti Code: its measurable properties include the concepts of *long methods*, *methods with no parameter*, *inheritance*; its lexical properties include the concepts of *procedural names*; its structural properties include the concepts of *global variables*, and *polymorphism*. The Spaghetti Code does not involve structural relationships among constituents. Such relationships appear in Blob and Functional Decomposition, for example through the key concepts *depends on data* and *associated with small classes*. Measurable properties are characterised by values specified using keywords such as *high, low, few,* and *many*, for example in the textual descriptions of the Blob, Functional Decomposition, and Swiss Army Knife, but not explicitly in the Spaghetti Code. The 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. More details on the properties and their possible values

The *Blob* (also called God class [14]) corresponds to a large controller class that ***depends on data stored*** in surrounding 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 [44]. 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***. ***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 I. List of Design Smells (The key concepts are in bold and italics.).

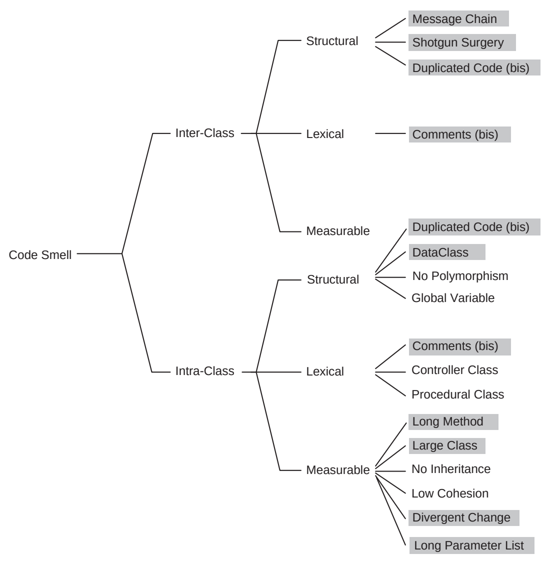

Fig. 2. Classification of some Code Smells. (Fowler's smells are in gray.)

for the key concepts are given in Section IV-B where we present the DSL built from the domain analysis, its grammar, and an exhaustive list of the properties and values.

Classification of Code Smells: Beck [2] provided a catalog of code smells but did not define any categories of or relationships among the smells. This lack of structuring hinders their identification, comparison, and, consequently, detection.

Efforts have been made to classify these symptoms. Mäntylä [15] proposed seven categories, such as object-orientation abusers or bloaters, including long methods, large classes, or long parameter lists. Wake [16] distinguished code smells that occur in or among classes. He further distinguished measurable smells, smells related to code duplication, smells due to conditional logic, and others. These two classifications are based on the nature of the smells. We are also interested

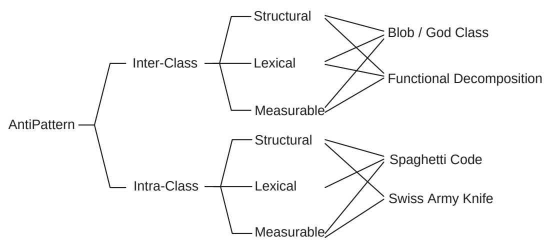

Fig. 3. Classification of Antipatterns.

in their properties, structure, and lexicon, as well as their coverage (intra- and inter-classes [45]) because these reflect better the spread of the smells.

Figure 2 shows the classification of some code smells. Following Wake, we distinguish code smells occurring in and among classes. We further divide the two subcategories into *structural*, *lexical*, and *measurable* code smells. This division helps in identifying appropriate detection techniques. For example, the detection of a structural smell may essentially be based on static analyses; the detection of a lexical smell may rely on natural language processing; the detection of a measurable smell may use metrics. Our classification is generic and classifies code smells in more than one category (*e.g.*, Duplicated Code).

Classification of Antipatterns: An antipattern [3] is a literary form describing a bad solution to a recurring design problem that has a negative impact on the quality of a system design. Contrary to design patterns, antipatterns describe what *not* to do. There exist general antipatterns [3] and antipatterns specific to concurrent processes [46], J2EE [47], [48], performance [49], XML [48], and other sub-fields of software engineering.

Brown *et al.* [3] classified antipatterns in three main categories: development, architecture, and project management. We focus on the antipatterns related to development and architecture because they represent poor design choices. Moreover, their correction may enhance the quality of systems and their detection is possible semi-automatically.

Figure 3 summarises the classification of the antipatterns. We use the previous classification of code smells to classify antipatterns according to their associated code smells. In particular, we distinguish between *intra-class* smells—smells in a class—and *inter-class* smells—smells spreading over more than one class. This distinction highlights the extent of the code inspection required to detect a smell. For example, we classify the Spaghetti Code antipattern as an intra-class design smell belonging to the structural, lexical, and measurable sub-categories because its code smells include long methods (measurable code smell), global variables (structural code smell), procedural names (lexical code smell), and absence of inheritance (another measurable code smell).

Taxonomy of Design Smells: Figure 4 summarises the classifications as a taxonomy in the form of a map. It is similar to Gamma *et al.*'s Pattern Map [1, inside back cover]. We only show the four design smells, including the Spaghetti Code, used in this paper for the sake of clarity.

This taxonomy describes the structural relationships among code and design smells, and their measurable, structural, and lexical properties (ovals in white). It also describes the structural relationships (edges) among design smells (hexagons) and some code smells (ovals in gray). It gives an overview of all key concepts that characterise a design smell. It also makes explicit the relationships among code and design smells.

Figure 4 presents the taxonomy that shows the relationships among design and code smells. This map is useful to prevent misinterpretation by clarifying and classifying smells based on their key concepts. Indeed, several sources of information may result in conflicting smell descriptions and the domain experts' judgement is required to resolve such conflicts. Lanza *et al.* [23] introduced the notion of *correlation webs* to also show the relationships among code smells. We introduce an additional level of granularity by adding antipatterns and include more information related to their properties.

3) Discussion:

The distinction between *structural* and *measurable* smells does not exclude the fact that the structure of a system is measurable. However, structural properties sometimes express better constraints among classes than metrics. While metrics report numbers, we may want to express the presence of a particular relation between two classes to describe a smell more precisely. In the example of the Spaghetti Code, we use a structural property to characterise polymorphism and a measurable property for inheritance. However, we could use a measurable property to characterise polymorphism and a structural property for inheritance. Such choices are left to domain experts who can choose the property that best fits their understanding of the smells in the context in which they want to detect them. With respect to the lexical properties, we use a list of keywords to identify specific names but, in future work, we plan to use WORDNET, a lexical database of English to deal with synonyms to widen the list of keywords.

The domain analysis is iterative because the addition of a new smell description may require the extraction of a new key concept, its comparison with existing concepts, and its classification. In our domain analysis, we study 29 smells including 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. These key concepts are at different levels of abstraction (structural relationships, properties, and values) and of different types (measurable, lexical, structural). They form a consistent vocabulary of reusable concepts to specify smells. In this step, we named the key concepts related to the Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. We will further detail these concepts in the next two steps.

Thus, our domain analysis is complete enough to describe a whole range of smells and can be extended, if required, during another iteration of the domain analysis. We have described without difficulty some new smells that were not used for the domain analysis. However, this domain analysis does not allow the description of smells related to the behavior of system. Current research work [50] will allow us to describe, specify, and detect this new category of smells.

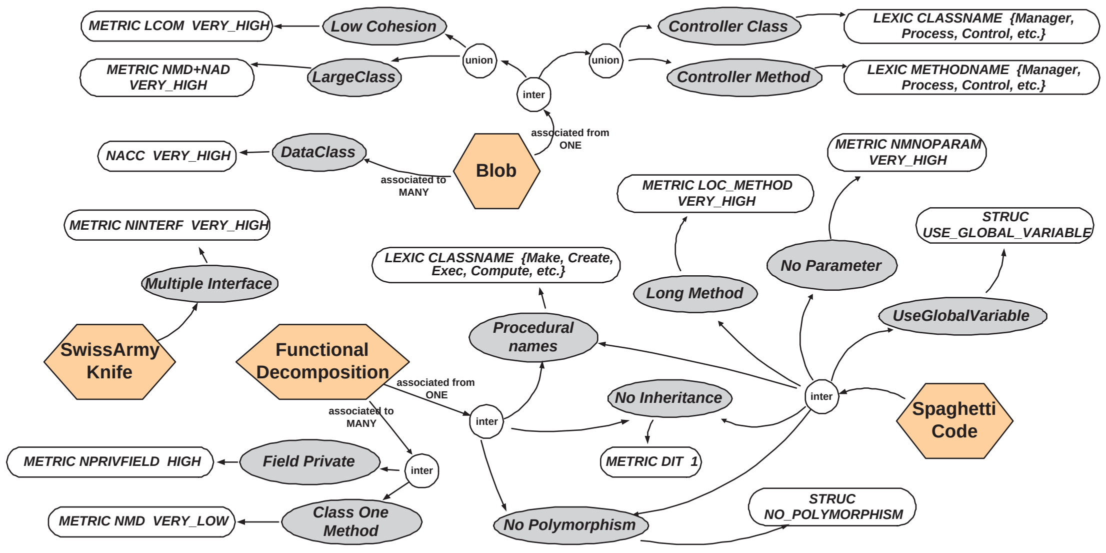

Fig. 4. Taxonomy of Smells. (Hexagons are antipatterns, gray ovals are code smells, and white ovals are properties.)

B. Step 2: Specification

1) Process:

Input: A vocabulary and taxonomy of smells.

Output: Specifications detailing the rules to apply on a model of a system to detect the specified smells.

Description: We formalise the concepts and properties required to specify detection rules at a high-level of abstraction using a DSL. The DSL allows the specification of smells in a declarative way as compositions of *rules* in *rule cards*. Using the smell vocabulary and taxonomy, we map rules with code smells and rules cards with design smells (*i.e.*, antipatterns). Each antipattern in the taxonomy corresponds to a rule card. Each code smell associated in the taxonomy with an antipattern is described as a rule. The properties in the taxonomy are directly used to express the rules. We make the choice of associating code smells with rules and antipatterns with rule cards for the sake of simplicity but without loss of generality for DETEX.

Implementation: Engineers manually define the specifications for the detection of smells using the taxonomy and vocabulary and, if needed, the context of the analysed systems.

As highlighted in the taxonomy, smells relate to the structure of classes (fields, methods) as well as to the structure of systems (classes and groups of related classes). For uniformity, we consider that smells characterise classes. Thus, a rule detecting long methods reports the *classes* defining these methods. A rule detecting the misuse of an association relationship returns the *class* at the source of the relationship. (It is also possible to obtain the class target of the relationship.) Thus, rules have a consistent granularity and their results can be combined using set operators. We chose class as level of granularity for the sake of simplicity and without loss of generality.

We define the DSL with a Backus Normal Form (BNF) grammar, shown in Figure 5. A rule card is identified by the keyword RULE_CARD, followed by a name and a set of rules specifying the design smell (line 1). A rule describes a list of properties, such as metrics (lines 8–11), relationships with other rules, such as associations (lines 14–16), and–or combination with other rules, based on available operators such as intersection or union (line 4). Properties can be of three different kinds: measurable, structural, or lexical, and define pairs of identifier–value (lines 5–7).

```
 1  rule_card     ::= RULE_CARD:rule_cardName { (rule)+ };
 2  rule          ::= RULE:ruleName { content_rule };

 3  content_rule  ::= operator ruleName (ruleName)+ | property | relationship
 4  operator      ::= INTER | UNION | DIFF | INCL | NEG

 5  property      ::= METRIC id_metric value_metric fuzziness
 6                  | LEXIC id_lexic ((lexic_value,)+)
 7                  | STRUCT id_struct
 8  id_metric     ::= DIT | NINTERF | NMNOPARAM | LCOM | LOC_CLASS
                    | LOC_METHOD | NAD | NMD | NACC | NPRIVFIELD
 9                  | id_metric + id_metric
10                  | id_metric - id_metric
11  value_metric  ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW
                    | NUMBER
12  id_lexic      ::= CLASS_NAME | INTERFACE_NAME | METHOD_NAME
                    | FIELD_NAME | PARAMETER_NAME
13  id_struct     ::= USE_GLOBAL_VARIABLE | NO_POLYMORPHISM
                    | IS_DATACLASS | ABSTRACT_CLASS
                    | ACCESSOR_METHOD | STATIC_METHOD
                    | FUNCTION_CLASS | FUNCTION_METHOD
                    | PROTECTED_METHOD | OVERRIDDEN_METHOD
                    | INHERITED_METHOD | INHERITED_VARIABLE

14  relationship  ::= rel_name FROM ruleName card TO ruleName card
15  rel_name      ::= ASSOC | AGGREG | COMPOS
16  card          ::= ONE | MANY | ONE_OR_MANY | OPTIONNALY_ONE

17   rule_cardName, ruleName, lexic_value ∈ string
18   fuzziness ∈ double
```

Fig. 5. BNF Grammar of Smell Rule Cards.

Measurable Properties: A measurable property defines a numerical or an ordinal value for a specific metric (lines 8–11). Ordinal values are defined with a five-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 the system under analysis. We define ordinal values with the box-plot statistical technique [51] to relate ordinal values with concrete metric values while avoiding setting artificial thresholds. Metric values can be added or subtracted. The degree of fuzziness defines the acceptable margin around the numerical value or around the threshold relative to the ordinal value (line 5). Although other tools implement the box-plot, such as iPLASMA [52], DETEX enhances this technique with fuzzy logic and thus alleviates the problem related to the definition of thresholds.

A set of metrics was identified during the domain analysis, including Chidamber and Kemerer metric suite [53], such as depth of inheritance `DIT`, lines of code in a class `LOC_CLASS`, lines of code in a method `LOC_METHOD`, number of attributes in a class `NAD`, number of methods `NMD`, lack of cohesion in methods `LCOM`, number of accessors `NACC`, number of private fields `NPRIVFIELD`, number of interfaces `NINTERF`, or number of methods with no parameters `NMNOPARAM`. The choice of the metrics is based on the taxonomy of the smells, which highlights the measurable properties needed to detect a given smell. This set of metrics is not restricted and can be easily extended with other metrics.

Lexical Properties: A lexical property relates to the vocabulary used to name a class, interface, method, field, or parameter (line 12). It characterises constituents with specific names defined in a list of keywords (line 6).

Structural Properties: A structural property relates to the structure of a constituent (class, interface, method, field, parameter, and so on) (lines 7, 13). For example, property `USE_GLOBAL_VARIABLE` checks that a class uses global variables while `NO_POLYMORPHISM` checks that a class that should use polymorphism does not. The BNF grammar specifies only a subset of possible structural properties, other can be added as new domain analyses are performed.

Set Operators: Properties can be combined using multiple set operators including intersection, union, difference, inclusion, and negation (line 4) (The negation represents the non-inclusion of one set in another).

Structural Relationships: System classes and interfaces characterised by the previous properties may also be linked with one another with different types of relationships including: association, aggregation, and composition [54] (lines 14–16). Cardinalities define the minimum and maximum numbers of instances of each class participating in a relationship.

2) Running Example:

Figure 6 shows the rule card of the Spaghetti Code, which characterises classes as Spaghetti Code using the intersection of six rules (line 2). A class is Spaghetti Code if it declares methods with a very high number of lines of code (measurable property, line 3), with no parameter (measurable property, line 4); if it does not use inheritance (measurable property, line 5), and polymorphism (structural property, line 6), and has a name that recalls procedural names (lexical property, line 7), while declaring/using global variables (structural property, line 8).

```
1  RULE_CARD:SpaghettiCode  {
2    RULE:SpaghettiCode
           { INTER LongMethod NoParamete NoInheritance
           NoPolymorphism ProceduralName UseGlobalVariable };
3    RULE:LongMethod        { METRIC LOC_METHOD VERY_HIGH 10.0 };
4    RULE:NoParameter       { METRIC NMNOPARAM VERY_HIGH 5.0 };
5    RULE:NoInheritance     { METRIC DIT 1 0.0 };
6    RULE:NoPolymorphism    { STRUCT NO_POLYMORPHISM };
7    RULE:ProceduralName    { LEXIC CLASS_NAME
                                 (Make, Create, Exec...) };
8    RULE:UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9  };
```

Fig. 6. Rule Card of the Spaghetti Code.

The Spaghetti Code does not include structural relationships because it is an intra-class defect. An example of such a relationship exists in the Blob where a large controller class must be associated with several data classes to be considered a Blob. Such a rule can be written as follows:

```
RULE:Blob { ASSOC FROM ControllerClass ONE TO DataClass MANY};
```

3) Discussion:

The domain analysis performed ensures that the specifications are built upon consistent high-level abstractions and capture domain expertise in contrast with general purpose languages, which are designed to be universal [55]. The DSL offers greater flexibility than ad-hoc detection algorithms. In particular, we made no reference at this point to the concrete implementation of the detection of the properties and structural relationships. Thus, it is easier for domain experts to understand the specifications because they are expressed using smell-related abstractions and they focus on *what* to detect instead of *how* to detect it, as in logic meta-programming [56]. Also, experts can modify easily the specifications at a high-level of abstraction without knowledge of the underlying detection framework, either by adding new rules or by modifying existing ones. They could for example use rule cards to specify smells dependent on industrial or technological contexts. For example, in small applications, they could consider as smells classes with a high `DIT` but not in large systems. In a management application, they could also consider different keywords as indicating controller classes.

The DSL is concise and expressive and provides a reasoning framework to specify meaningful rules. Moreover, we wanted to avoid an imperative language where, for example, we would use a rule like `method[1].parameters.size = 0` to obtain classes with methods with no parameters. Indeed, using the DSL should not require computer skills or knowledge about the underlying framework or meta-model, to be accessible to most experts. In our experiments, graduate students wrote specifications in less than 15 minutes, depending on their familiarity with the smells, with no knowledge of the underlying framework. We provide some rule cards at [57].

Since the method is iterative, if a key concept is missed, we can add it to the DSL later. The method as well as the language are flexible. The flexibility of the rule cards depends on the expressiveness of the language and available key concepts, which has been tested on a representative set of smells, eight antipatterns and 21 code smells.

C. Step 3: Generation of the Algorithms

We briefly present here the generation step of algorithms for the sake of completeness; details are available in [7].

1) Process:

Input: Rule cards of smells.

Output: Detection algorithms for the smells.

Description: We reify the smell specifications to allow algorithms to access and manipulate programmatically the resulting models. Reification is an important mechanism to manipulate concepts programmatically [58]. From the DSL, we build a meta-model, SMELLDL (*Smell Definition Language*), and a parser to model rule cards and manipulate these SMELLDL models programmatically. Then, we automatically generate algorithms using templates. The detection algorithms are based both on the models of the smells and on models of systems. The generated detection algorithms are correct by construction of our specifications using a DSL [59].

Implementation: The reification is automatic using the parser with the SMELLDL meta-model. The generation is also automatic and relies on our SMELLFW (*Smell FrameWork*) framework, which provides services common to all detection algorithms. These services implement operations on the relationships, operators, properties, and ordinal values. The framework also provides services to build, access, and analyse system models. Thus, we can 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 have been directed by the key concepts from the domain analysis and the DSL.

Meta-model of Rule Cards: Figure 7 is an excerpt of the SMELLDL meta-model, which defines constituents to represent rule cards, rules, set operators, relationships among rules, and properties. A rule card is specified concretely as an instance of class `RuleCard`. An instance of `RuleCard` is composed of objects of type `IRule`, which describes rules that can be either simple or composite. A composite rule, `CompositeRule`, is composed of other rules, using the Composite design pattern [1]. Rules are combined using set operators defined in class `Operators`. Structural relationships are enforced using methods in class `Relationships`. The meta-model also implements the Visitor design pattern. A parser analyses the rule cards and produces an instance of class `RuleCard`. The parser is built using JFLEX and JAVACUP and the BNF grammar shown in Figure 5.

Framework for Detection: The SMELLFW framework is built upon the PADL meta-model (*Pattern and Abstract-level Description Language*) [12] and on the POM framework (*Primitives, Operators, Metrics*) for metric computation [60]. PADL is a language-independent meta-model to represent object-oriented systems [61], including binary class relationships [54] and accessors. PADL offers a set of constituents (classes, interfaces, methods, fields, relationships…) to build models of systems. It also provides methods to manipulate

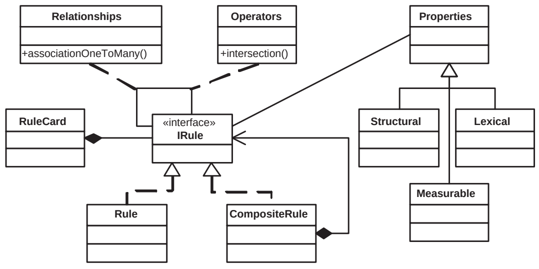

Fig. 7. Meta-model SMELLDL.

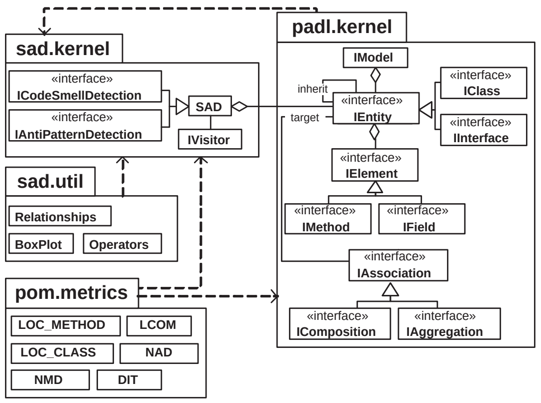

Fig. 8. Architecture of the SMELLFW Framework.

these models and generate other models, using the Visitor design pattern. We choose PADL because it has six years of active development and is maintained in-house. We could have used another meta-model such as FAMOOS [62] or GXL [63], or a source model extractor, such as LSME [64].

Figure 8 sketches the architecture of the SMELLFW framework, which consists of two main packages, `sad.kernel` and `sad.util`. Package `sad.kernel` contains core classes and interfaces. Class `SAD` represents smells and is so far specialised in two subclasses, `AntiPattern` 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 `IAntiPatternDetection` and `ICodeSmellDetection` define the services that detection algorithms must provide. Package `sad.util` declares utility classes that allow the manipulation of some key concepts of the rule cards.

Set Operators. Class `Operators` package `sad.util` defines the methods required to perform intersection, union, difference, inclusion, and negation between code smells. These operators work on the sets of classes that are potential code smells. They return new sets containing only the appropriate classes. For example, the code below performs an intersection on the set of classes that contain methods without parameter and those with long methods:

```
final Set setOfLongMethodsWithNoParameter =
  CodeSmellOperators.intersection(
      setOfLongMethods,
      setOfMethodsWithNoParameter);
```

Measurable Properties. Properties based on metrics are computed using POM, which provides 44 metrics, such as lines of code in a class `LOC_CLASS`, number of declared methods `NMD`, or lack of cohesion in methods `LCOM`, and is easily extensible. Using POM, SMELLFW can compute any metric on a set of classes. For example, in the code below, the metric `LOC_CLASS` is computed on each class of a system:

```
final IClass aClass = iteratorOnClasses.next();
final double aClassLOC =
  Metrics.compute("LOC_CLASS", aClass);
```

Class `BoxPlot` in package `sad.util` offer methods to computes and access the quartiles for and outliers of a set of metric values as illustrated in the following code excerpt:

```
double fuzziness = 0.1;
final BoxPlot boxPlot =
  new BoxPlot(LOCofSetOfClasses, fuzziness);
final Map setOfOutliers = boxPlot.getHighOutliers();
```

Lexical Property. The verification of lexical properties stems from PADL, which checks the names of classes, methods, and fields against names defined in the rule cards. The following code checks, for each class of a system, if its name contains one of the strings specified in a predefined list:

```
String[] CTRL_NAMES =
  new String[] { "Calculate", "Display", ..., "Make" };

final IClass aClass = iteratorOnClasses.next();
for (int i = 0; i < CTRL_NAMES.length; i++) {
  if (aClass.getName().contains(CTRL_NAMES[i])) {
      // do something
  }
}
```

Structural Properties. Any structural property can be verified using PADL, which provides all the constituents and methods to assess structural properties. For example, the method `isAbstract()` returns true if a class is abstract:

```
final IClass aClass = iteratorOnClasses.next();
boolean isClassAbstract = aClass.isAbstract();
```

Structural Relationships. PADL also provides constituents describing binary class relationships. We can enforce the existence of certain relationships among classes being potentially a smell, *e.g.*, an association between a main class and its data classes as illustrated by the following code excerpt:

```
final Set setOfCandidateBlobs =
  Relations.associationOneToMany(setOfMainClasses,
                                 setOfDataClasses);
```

Algorithm Generation: An instance of class `RuleCard` is the entry point to a model of a rule card. The generation of the detection algorithms is implemented as a visitor on models of rule cards that generates the appropriate source code, based on templates and the services provided by SMELLFW, as illustrated in the following running example. Templates are excerpts of JAVA source code with well-defined tags to be replaced by concrete code. More details on the templates and generation algorithm can be found in [7].

2) Running Example:

The following code excerpt presents the visit method that generates the detection rule associated to a measurable property. When a model of the rule is visited, tag `<CODESMELL>` is replaced by the name of the rule, tag `<METRIC>` by the name of the metric, tag `<FUZZINESS>` by the associated value of the fuzziness in the rule, and tag `<ORDINAL_VALUES>` by the method associated with the ordinal value.

```
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) {
  String method = null;
  switch (value) {
    case VERY_HIGH : method = "getHighOutliers";
    break;
    case HIGH      : method = "getHighValues";
    break;
    case MEDIUM    : method = "getNormalValues";
    break;
    default        : method = "getNormalValues";
    break;
  }
  return method;
}
```

The detection algorithm for a design defect is declared as implementing interface `IAntiPatternDetection`. The algorithm aggregates the detection algorithms of several code smells, implementing interface `ICodeSmellDetection`. The results of the detections of code smells are combined using set operators to obtain suspicious classes for the antipattern. Excerpts of generated Spaghetti Code detection algorithm can be found in [7] and on the companion Web site [57].

3) Discussion:

The SMELLDL meta-model and the SMELLFW framework, along with the PADL meta-model and the POM framework, provide the concrete mechanisms to generate and apply detection algorithms. However, using DECOR we could design another language and build another meta-model with the same capabilities. Detection algorithms could be generated against other frameworks. In particular, we could reuse some of the tools presented in the related work in Section II-C.

The addition of another property in the DSL requires the implementation of the analysis within SMELLFW. We experimented informally with the addition of new properties and it took from 15 minutes to one day to add a new property, depending on the complexity of the analysis. This operation is necessary only once per new property.

SMELLDL models must be instantiated for each smell but the SMELLDL meta-model and the SMELLFW framework are generic and do not need to be redefined. Models of systems are built before applying the detection algorithms, while metric values are computed on the fly and as needed.

D. Step 4: Detection

1) Process:

Input: Smell detection algorithms and the model of a system in which to detect the smells.

Output: Suspicious classes whose properties and relationships conform to the smells specifications.

Description: We automatically apply the detection algorithms on models of systems to detect suspicious classes. Detection algorithms may be applied in isolation or in batch.

Implementation: Calling the generated detection algorithms is straightforward, using the services provided by SMELLFW. The model of a system could be obtained using reverse engineering by instantiating the constituents of PADL, sketched in Figure 8, or from design documents.

2) Running Example:

Following our running example of the Spaghetti Code and XERCES v2.7.0, we first obtain a model of XERCES, based on the constituents of PADL. We then apply the detection algorithm of the Spaghetti Code on this model to detect and report suspicious classes, using the code exemplified below. In XERCES v2.7.0, we found 76 suspicious Spaghetti Code classes among the 513 classes of the system.

```
IAntiPatternDetection antiPatternDetection =
  new SpaghettiCodeDetection(model);
antiPatternDetection.performDetection();
...
outputFile.println(
  antiPatternDetection.getSetOfAntiPatterns());
```

3) Discussion:

Models on which the detection algorithms are applied can be obtained from original designs produced during forward or from reverse engineering, because industrial designs are seldom available freely. Also, design documents, like documentation in general, are often out-of-date. In many systems with poor documentation, the source code is the only reliable source of information [65] that it is precise and up-to-date. Thus, because the efficiency of the detection depends on the model of the system, we chose to work with reverse-engineered data, which provides richer data than usual class diagrams, for example method invocations. DETEX would also apply to class diagrams, yet certain rules would no longer be valid. Thus, we did not analyse class diagrams directly and let such a study a future work.

V. VALIDATION

Previous detection approaches have been validated on few smells and proprietary systems. Thus, as our third contribution, in addition to the DECOR method and DETEX detection technique, we validate DETEX. The aim of this validation is to study both the application of the four steps of DETEX and the results of their application using four design smells, their 15 code smells, and 11 open-source systems. The validation is performed by independent engineers who assess whether suspicious classes are smells, depending on the contexts of the systems. We put aside domain analysis and smell specification because these steps are manual and their iterative processes would be lengthy to describe.

A. Assumptions of the Validation

We want to validate the three following assumptions:

1) *The DSL allows the specification of many different smells.* This assumption supports the applicability of DETEX on four design smells, composed of 15 code smells, and the consistency of the specifications.
2) *The generated detection algorithms have a recall of 100%, i.e., all known design smells are detected, and a precision greater than 50%, i.e., the detection algorithms are better than random chance.* Given the trade-off between precision and recall, we assume that 50% precision is significant enough with respect to 100% recall. This assumption supports the precision of the rule cards and the adequacy of the algorithm generation and of the SMELLFW framework.
3) *The complexity of the generated algorithms is reasonable, i.e., computation times are in the order of one minute.* This assumption supports the precision of the generated algorithms and the performance of the services of the SMELLFW framework.

B. Subjects of the Validation

We use DETEX to describe four well-known but different antipatterns from Brown [3]: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. Table I summarises these smells, which include in their specifications 15 different code smells, some of which described in Fowler [2]. We automatically generate associated detection algorithms.

C. Process of the Validation

We validate the results of the detection algorithms by analysing the suspicious classes manually to (1) validate suspicious classes as true positives in the context of the systems and (2) identify false negatives, *i.e.*, smells not reported by our algorithms. Thus, we recast our work in the domain of information retrieval to use the measures of precision and recall [66]. Precision assesses the number of true smells identified among the detected smells, while recall assesses the number of detected smells among the existing smells:

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

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

We asked independent engineers to compute the recall of the generated algorithms. Validation is performed manually because only engineers can assess whether a suspicious class is indeed a smell or a false positive, depending on the smell descriptions and the systems' contexts and characteristics. This step is time consuming if the smell specifications are not restrictive enough and the number of suspected classes is large.

D. Objects of the Validation

We perform the validation using the reverse-engineered models of ten open-source JAVA systems: ARGOUML, AZUREUS, GANTTPROJECT, LOG4J, LUCENE, NUTCH, PMD, QUICKUML, and two versions of XERCES. In contrast to previous work, we use freely available systems to ease comparisons and replications. We provide some information on these systems in Table II. We also apply the algorithms on ECLIPSE but only discuss their results.

Name	Version	Lines of Code	Number of Classes	Number of Interfaces
ARGOUML	0.19.8	113,017	1,230	67
An extensive UML modelling tool				
AZUREUS	2.3.0.6	191,963	1,449	546
A peer-to-peer client implementing the BitTorrent protocol				
GANTTPROJECT	1.10.2	21,267	188	41
A project-management tool to plan projects with Gantt charts				
LOG4J	1.2.1	10,224	189	14
A logging JAVA package				
LUCENE	1.4	10,614	154	14
A full-featured text-search JAVA engine				
NUTCH	0.7.1	19,123	207	40
An open-source web search engine, based on LUCENE				
PMD	1.8	41,554	423	23
A JAVA source code analyser for identifying low-level problems				
QUICKUML	2001	9,210	142	13
A simple UML class and sequence diagrams modelling tool				
XERCES	1.0.1	27,903	189	107
A framework for building XML parsers in JAVA				
XERCES	2.7.0	71,217	513	162
Release of March 2006 of the XERCES XML parser				

TABLE II. List of Systems.

E. Results of the Validation

We report results in three steps. First, we report the precisions and recalls of the detection algorithms for XERCES v2.7.0 for the four design smells using data obtained independently. These data constitute the first available report on the precision and recall of a detection technique. Then, we report the precisions and computation times of the detection algorithms on the ten reverse-engineered open-source systems to show the scalability of DETEX. We illustrate these results by concrete examples. Finally, we also apply our detection algorithms on ECLIPSE v3.1.2, demonstrating their scalability and highlighting the problem of balance among numbers of suspicious classes, precisions, and system context.

1) Precision and Recall on XERCES*:*

We asked three master's students and two independent engineers to manually analyse XERCES v2.7.0 using only Brown's and Fowler's books as references. They used an integrated development environment, ECLIPSE, to visualise the source code and studied each class separately. When in doubt, they referred to the books and decided by consensus using a majority vote whether a class was actually a design smell. They performed a thorough study of XERCES and produced a XML file containing suspicious classes for the four design smells. A few design smells may have been missed by mistake due to the nature of the task. We will ask as future work other engineers to perform this same task again to confirm the findings and on other systems to increase our database.

Table III presents the precision and recall of the detection of the four design smells in XERCES v2.7.0. We perform all computations on an Intel Dual Core at 1.67GHz with 1Gb of RAM. Computation times do not include building the system model but include computing metrics and checking structural relationships and lexical and structural properties.

Smells	Numbers of True Positives		Numbers of Detected Smells		Precision	Recall	Time
Blob	39/513	(7.6%)	44/513	(8.6%)	88.6%	100%	2.45s
F.D.	15/513	(3.0%)	29/513	(5.6%)	51.7%	100%	0.16s
S.C.	46/513	(9.0%)	76/513	(15%)	60.5%	100%	0.22s
S.A.K.	23/513	(4.5%)	56/513	(11%)	41.1%	100%	0.05s
					60.5%	100%	0.72s

TABLE III. Precision and Recall in XERCES v2.7.0, which contains 513 classes. (F.D. = Functional Decomposition, S.C. = Spaghetti Code, and S.A.K. = Swiss Army Knife).

The recalls of our detection algorithms are 100% for each design smell. We specified the detection rules to obtain a perfect recall and assess its impact on precision. Precision is between 41.1% and close to 90% (with an overall precision of 60.5%), providing between 5.6% and 15% of the total number of classes, which is reasonable to analyse manually, compared with analysing the entire system of 513 classes. These results also provide a basis for comparison with other approaches.

2) Running Example:

We found 76 suspicious classes for the detection of the Spaghetti Code design smell in XERCES v2.7.0. Out of these 76 suspicious classes, 46 are indeed Spaghetti Code previously identified in XERCES manually by engineers independent of the authors, which leads to a precision of 60.5% and a recall of 100% (see third line in Table III).

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, because it does not use inheritance and polymorphism but uses excessively global variables. Moreover, this method weighs 759 LOC, while the upper method length computed using the box-plot is 254.5 LOC. The result file is illustrated below:

```
1.Name = SpaghettiCode
1.Class = org.apache.xerces.xinclude.XIncludeHandler
1.NoInheritance.DIT-0 = 1.0
1.LongMethod.Name = handleIncludeElement(XMLAttributes)
1.LongMethod.LOC_METHOD = 759.0
1.LongMethod.LOC_METHOD_Max = 254.5
1.GlobalVariable-0 = SYMBOL_TABLE
1.GlobalVariable-1 = ERROR_REPORTER
1.GlobalVariable-2 = ENTITY_RESOLVER
1.GlobalVariable-3 = BUFFER_SIZE
1.GlobalVariable-4 = PARSER_SETTINGS

2.Name = SpaghettiCode
2.Class = org.apache.xerces.impl.xpath.regex.RegularExpression
2.NoInheritance.DIT-0 = 1.0
2.LongMethod.Name = matchCharArray(Context,Op,int,int,int)
2.LongMethod.LOC_METHOD = 1246.0
2.LongMethod.LOC_METHOD_Max = 254.5
2.GlobalVariable-0 = WT_OTHER
2.GlobalVariable-1 = WT_IGNORE
2.GlobalVariable-2 = EXTENDED_COMMENT
2.GlobalVariable-3 = CARRIAGE_RETURN
2.GlobalVariable-4 = IGNORE_CASE
...
```

Another example is class `org.apache.xerces.impl.xpath.regex.RegularExpression` declaring method `matchCharArray(Context,Op,int,int,int)` with a size of 1,246 LOC. Looking at the code, we see that this method contains a switch statement and duplicated code for 20 different operators (such as =, $<$, $>$, [a-z]...) while class `org.apache.xerces.impl.xpath.regex.Op` actually has subclasses for most of these operators. This method could have been implemented in a more object-oriented style by dispatching the matching operator to `Op` subclasses to split the large method into smaller ones in the subclasses. However, such design would introduce polymorphic calls into the method traversing all characters of an array. Therefore, XERCES designers may not have opt for such a design to optimize performance at the cost of maintainability.

The 46 Spaghetti Code represent true positives and include "bad" Spaghetti Code such as method `handleIncludeElement` but also "good" Spaghetti Code such as method `matchCharArray`. The "good" smells were not rejected because they could represent weak spots in terms of quality and maintenance. Other examples of typical Spaghetti Code detected and checked as true positives are classes generated automatically by parser generators. The 30 other suspicious classes were rejected by the independent engineers and are false positives. Even if these classes verified the characteristics of Spaghetti Code, most of them were easy to understand, and thus, were considered false positives. Thus, it would be necessary to add other rules or modify the existing ones to narrow the set of candidate classes, for example, by detecting nested `if` statements and loops, characterising complex code.

3) *Results on Other Systems:*

Table 9 provides for the nine other systems plus XERCES v2.7.0 the numbers of suspicious classes in the first line of each row; the numbers of true design smells in the second line; the precisions in the third; and the computation times in the fourth. We only report precisions: recalls on other systems than XERCES are future work due to the required time-consuming manual analyses. We have also performed all computations on an Intel Dual Core at 1.67GHz with 1Gb of RAM.

4) *Illustrations of the Results:*

We briefly present examples of the four design smells. In XERCES, method `handleIncludeElement (XMLAttributes)` of the `org.apache.xerces. xinclude.XIncludeHandler` class is a typical example of Spaghetti Code. A good example of Blob is class `com.aelitis.azureus.core.dht.control.impl. DHTControlImpl` in AZUREUS. This class declares 54 fields and 80 methods for 2,965 lines of code. An interesting example of Functional Decomposition is class `org.argouml.uml.cognitive.critics.Init` in ARGOUML, in particular because the name of the class includes a suspicious term, `init` that suggests a functional programming. Class `org.apache.xerces.impl.dtd. DTDGrammar` is a striking example of Swiss Army Knife in XERCES, implementing four different sets of services with 71 fields and 93 methods for 1,146 lines of code.

5) *Results on* ECLIPSE *for the Scalability:*

We also apply our detection algorithms on ECLIPSE to demonstrate their scalability. ECLIPSE v3.1.2 weighs 2,538,774 lines of code for 9,099 classes and 1,850 interfaces. It is one order of magnitude larger than the largest of the open-source systems, AZUREUS. The detection of the four design smells in ECLIPSE requires more time and produces more results. We detect 848, 608, 436, and 520 suspicious classes for the Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife design smells, respectively. The detections take about 1h20m for each smell, with another hour to build the model. The use of the detection algorithms on ECLIPSE shows the scalability of our implementation. It also highlights the balance between numbers of suspicious classes and precisions. Indeed, if the choice is to maximise recall, the number of suspicious classes may be high, even more so in large systems, and thus precision will be low. Conversely, if the choice is to minimise the number of suspicious classes, precision will be high but recall may be low. In addition, it shows the importance of specifying smells in the context of the system in which they are detected. Indeed, the large number of suspicious classes for Blob in ECLIPSE, about $1/10^{th}$ of the overall number of classes, may come from design and implementation choices and constraints within the ECLIPSE community and thus, the smell specifications should be adapted to consider these choices. With our method and detection technique, engineers can easily re-specify smells to fit their context and environment and get greater precision.

F. Discussion of the Results

We verify each of the three assumptions using the results of the validation of DETEX.

1) *The DSL allows the specification of many different smells.* We described four different design smells of inter- and intra-class categories and of the structural, lexical, and measurable categories, as shown in Figure 3. These four smells are characterised by 15 code smells also belonging to 6 different categories, shown in Figure 2. Thus, we showed that we can describe many different smells, which supports the efficiency of our detection technique and the generality of its DSL.
2) *The generated detection algorithms have a recall of 100% and a precision greater than 50%.* Table III shows that the precision and recall for XERCES v2.7.0 fulfill our assumptions with a precision of 60.5% and a recall of 100%. Table 9 presents the precisions for the other nine systems, which almost all comply with our assumption, with a precision greater than 50% (except for two systems), thus validating the usefulness of our detection technique.
3) *The complexity of the generated algorithms is reasonable, i.e., computation times are in the order of one minute.* Computation times are in general less than a few seconds (except for ECLIPSE which took about 1 hour) because the complexity of the detection algorithms depends only on the number of classes in a system, n, and on the number of properties to verify on each class: $(c + op) \times \mathcal{O}(n)$, where c is the number of properties and op the number of operators.

The computation times of the design smells vary with the smells and the systems. During validation, we noticed that building the models of the systems took up most of the computation times, while the detection algorithms have

	ArgoUML	Azureus	GanttProject	Log4J	Lucene	Nutch	PMD	QuickUML	Xerces v1.0.1	Xerces v2.7.0
Blob	29 (2.4%)	41 (2.8%)	10 (5.3%)	3 (1.6%)	3 (1.9%)	6 (2.9%)	4 (0.9%)	0 (0%)	10 (5.3%)	44 (8.6%)
	25 (2.0%)	38 (2.6%)	9 (4.8%)	3 (1.6%)	2 (1.3%)	4 (1.9%)	4 (0.9%)	0 (0%)	10 (5.3%)	39 (7.6%)
	86.2%	92.7%	90.0%	100%	66.7%	66.7%	100%	100%	100%	88.6%
	3.0s	6.4s	2.4s	1.3s	1.8s	3.6s	3.9s	0.4s	2.7s	2.4s
F.D.	37 (3.0%)	44 (3.0%)	15 (8.0%)	11 (5.8%)	1 (0.6%)	15 (7.2%)	13 (3.1%)	10 (7.0%)	4 (2.1%)	29 (5.6%)
	22 (1.8%)	17 (1.2%)	4 (2.1%)	6 (3.2%)	0 (0%)	3 (1.4%)	4 (0.9%)	3 (2.1%)	4 (2.1%)	15 (2.9%)
	59.5%	38.6%	26.7%	54.5%	0%	20.0%	30.8%	30.0%	100%	51.7%
	0.4s	0.5s	0.8s	0.05s	0.03s	0.05s	0.06s	0.02s	0.03s	0.16s
S.C.	44 (3.6%)	153 (15.6%)	14 (7.4%)	3 (1.6%)	8 (5.2%)	26 (12.6%)	9 (2.1%)	5 (3.5%)	25 (13.2%)	76 (14.8%)
	38 (3.1%)	125 (8.6%)	10 (5.3%)	2 (1.1%)	6 (3.9%)	22 (10.6%)	5 (1.2%)	0 (0%)	23 (12.2%)	46 (9.0%)
	86.4%	81.7%	71.4%	66.7%	75.0%	84.6%	55.6%	0%	92.0%	60.5%
	0.3s	2.9s	0.2s	0.08s	0.09s	0.1s	0.06s	0.03s	0.11s	0.2s
S.A.K.	108 (8.8%)	145 (10.0%)	8 (4.2%)	51 (27.0%)	9 (5.8%)	33 (15.9%)	13 (3.1%)	6 (4.2%)	12 (6.3%)	56 (10.9%)
	18 (1.5%)	33 (2.3%)	3 (1.6%)	33 (17.5%)	1 (0.6%)	13 (6.3%)	6 (1.4%)	1 (0.7%)	5 (2.6%)	23 (4.5%)
	16.6%	22.7%	37.5%	64.7%	11.1%	39.4%	46.1%	16.7%	41.7%	41.1%
	0.3s	0.13s	0.05s	0.02s	0.02s	0.02s	0.02s	0.02s	0.03s	0.05s
	62.2%	58.9%	56.4%	71.5%	38.2%	52.7%	58.1%	36.7%	83.4%	60.5%

Fig. 9. Results of Applying the Detection Algorithms. (In each row, the first line is the number of suspicious classes, the second line is the number of classes *being* design smells, the third line is the precision, and the fourth line shows the computation time. Numbers in parenthesis are the percentages of the classes being reported. The last row corresponds to the average precision per system. (F.D. = Functional Decomposition, S.C. = Spaghetti Code, and S.A.K. = Swiss Army Knife))

short execution times, which explains the minor differences between each system, in the same line in Table 9, and the differences between each design smell, in different columns. The computation times for PADL models are not surprising because the models contain extensive data, including binary class relationships [54] and accessors.

The precisions also vary in relation to the design smells and the systems, as shown in Table 9: First, the systems have been developed in different contexts and may have unequal quality. Systems such as AZUREUS or XERCES may be of lesser quality than LUCENE or QUICKUML, thus leading to greater numbers of suspicious classes that are actually smells. However, the low number of smells detected in LUCENE and QUICKUML leads to a low precision. For example, only one Functional Decomposition was detected in LUCENE, but it was a false positive, thus leading to a precision of 0% and an average precision of 38.2%. The smell specifications can be over- or under-constraining. For example, the rule cards of the Blob and Spaghetti Code specify the smells strictly using metrics and structural relationships, leading to a low number of suspicious classes and high precisions. The rule cards of the Functional Decomposition and Swiss Army Knife specify these smells loosely using lexical data, leading to lower precisions. Thus, the specifications must not be too loose, not to detect too many suspicious classes, or too restrictive, to miss smells. With DETEX, engineers can refine the specifications systematically, according to the detected suspicious classes and their knowledge of the systems. The choice of metrics and thresholds is left to the domain experts to take into account the context and characteristics of the analysed systems.

The number of false positives appears quite high; however, we obtained many false positives because our objective was 100% recall for all systems. Using DETEX and its DSL, the rules can be refined systematically and easily to fit the specific contexts of the analysed systems and thus to increase precisions if desired, possibly at the expense of recall. Thus, the number of false positives will be low and engineers will not spend time checking a vast amount of false results. As future work, we propose to sort the results in critical order, *i.e.*, according to the classes that are the most likely to be smells, to help engineers in assessing the results. The numbers of suspicious classes obtained are usually orders of magnitude lower than the overall number of classes in a system; thus, the detection technique indeed ease engineers' code inspection.

We also *indirectly* validated the usefulness of DECOR by validating DETEX. Indeed, DECOR is the method of which *one* instantiation is DETEX. Therefore, the validation of DETEX showed that the DECOR method provides the necessary steps from which to derive a valid detection technique. As a metaphor, we could assimilate DECOR to a class and DETEX to one of its instances that has been successfully tested, thus showing the soundness of its class.

G. *Threats to Validity*

Internal Validity: The obtained results depend on the services provided by the SMELLFW framework. Our current implementation allows the detection of classes that strictly conform to the rule cards and we only handle a degree of fuzziness in measurable properties. This choice of implementation does not limit DETEX intrinsically because it could accommodate other implementations of its underlying detection framework. The results also depend on the specifications of the design smells. Thus, we used for the experiments a representative set of smells so as not to influence the results.

External Validity: One threat to the validity of the validation is the exclusive use of open-source JAVA systems. The open-source development process may bias the number of design smells, especially in the case of mature systems such as PMD v1.8 or XERCES v2.7.0. Also, using JAVA may impact design and implementation choices and thus the presence of smells. However, we applied our algorithms on systems of various sizes and qualities to preclude the possibility for all systems to be either well or badly implemented. Moreover, we performed a validation on open-source systems to allow comparisons and replications. We are in contact with software companies to replicate this validation on their proprietary systems.

Construct Validity: The subjective nature of identifying or specifying smells and assessing suspicious classes as smells is a threat to construct validity. Indeed, our understanding of smells may differ from that of other engineers. We lessen this threat by specifying smells based on general literature and drawing inspiration from previous work. We also asked the engineers in charge of computing precision and recall to do so. Moreover, we contacted developers involved in each of the analysed systems to validate our results and to improve our smell specifications. So far, we have received a few answers but enthusiastic interest. Engineers analysed independently our results for LOG4J, LUCENE, PMD, and QUICKUML, and confirmed the results in Table 9. We thank M. Adamovic, C. Alphonce, D. Cutting, T. Copeland, P. Gardner, E. Ross, and Y. Shapira for their kind help. We are in the process of increasing the size of our library of smells thanks to their support. We believe important to report the detection results to the communities developing the systems.

Repeatability/Reliability Validity: The results of the validation are repeatable and reliable because we use freely open-source programs that can be freely downloaded from the Internet. Also, our implementation is available upon request while all its results are on the companion Web site [57].

VI. Conclusion and Future Work

The detection of smells is important to improve the quality of software systems, to facilitate their evolution, and thus to reduce the overall cost of their development and maintenance.

We proposed the following improvements to previous work. First, we introduced DECOR, a method that embodies all the step necessary to define detection techniques. Second, we cast our detection technique, now called DETEX, in the context of the DECOR method. DETEX now plays the role of reference instantiation of our method. It is supported by a DSL for specifying smells using high-level abstractions, taking into account the context of the analysed systems, and resulting from a thorough domain analysis of the text-based descriptions of the smells. Third, we applied DETEX on four design smells and their 15 underlying code smells and discussed its usefulness, precision, and recall. This is the first such extensive validation of a smell detection technique.

Our detection technique and the inputs, outputs, processes, and implementations defined in each step can be generalised to other smells. Also, it can be implemented using other techniques as long as they provide relevant data for the considered steps. We have not compared our implementation with other approaches but will do so in future work.

Future work includes using the WORDNET dictionary; using existing tools to improve the implementation of our method; improving the quality and performance of the source code of the generated detection algorithms; computing the recall on other systems; applying our detection technique to other kinds of smells; comparing quantitatively our method with previous work. With respect to the last work, we are currently conducting a study on smells detection tools including several tools such as RevJava, FindBugs, PMD, Hammurapi, or Lint4j to our detection technique against existing tools. A first comparison is available in the related work.

Acknowledgments: We are grateful to G. Antoniol, K. Mens, and D. Thomas for their comments on earlier versions of this paper. We thank M. Amine El Haimer and N. Tajeddine for applying the method and detection technique on several smells. We thank D. Huynh and P. Leduc for their help with the implementation of parts of the SMELLFW framework. Finally, we express our gratitude to the developers who confirmed our findings in the open-source systems.

Y.-G. Guéhéneuc was partially supported by a NSERC Discovery Grant. N. Moha was supported by the Université de Montréal and The FQRNT (Fonds québécois de la recherche sur la nature et les technologies), a funding agency of the Gouvernement du Québec.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

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

[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*, 1^{st} ed. John Wiley and Sons, March 1998.

[4] R. S. Pressman, *Software Engineering – A Practitioner's Approach*, 5^{th} ed. McGraw-Hill Higher Education, November 2001.

[5] 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} Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.

[6] N. Moha, Y.-G. Guéhéneuc, and P. Leduc, "Automatic generation of detection algorithms for design defects," in *Proceedings of the 21^{st} Conference on Automated Software Engineering*, S. Uchitel and S. Easterbrook, Eds. IEEE Computer Society Press, September 2006, pp. 297–300, short paper.

[7] N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, and L. Duchien, "A domain analysis to specify design defects and generate detection algorithms," in *Proceedings of the 11^{th} international conference on Fundamental Approaches to Software Engineering*, J. Fiadeiro and P. Inverardi, Eds. Springer-Verlag, March-April 2008.

[8] B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[9] G. Bruno, P. Garza, E. Quintarelli, and R. Rossato, "Anomaly detection in xml databases by means of association rules," in *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 387–391.

[10] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the detection of snapshot isolation anomalies," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 1263–1274.

[11] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Comput. Netw.*, vol. 51, no. 12, pp. 3448–3470, 2007.

[12] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, September 2008.

[13] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

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

[15] M. Mantyla, "Bad smells in software - a taxonomy and an empirical study." Ph.D. dissertation, Helsinki University of Technology, 2003.

[16] W. C. Wake, *Refactoring Workbook*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

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

[18] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the* 11^{th} *International Software Metrics Symposium*, F. Lanubile and C. Seaman, Eds. IEEE Computer Society Press, September 2005.

[19] E. H. Alikacem and H. Sahraoui, "Generic metric extraction framework," in *Proceedings of the* 16^{th} *International Workshop on Software Measurement and Metrik Kongress (IWSM/MetriKon)*, 2006, pp. 383–390.

[20] K. Dhambri, H. Sahraoui, and P. Poulin, "Visual detection of design anomalies." in *Proceedings of the* 12^{th} *European Conference on Software Maintenance and Reengineering, Tampere, Finland*. IEEE Computer Society, April 2008, pp. 279–283.

[21] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 30.

[22] G. Langelier, H. A. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the* 20^{th} *International Conference on Automated Software Engineering*, T. Ellman and A. Zisma, Eds. ACM Press, November 2005.

[23] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[24] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society Press, Oct. 2002.

[25] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Software*, vol. 12, no. 6, pp. 17–26, 1995.

[26] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

[27] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 2, pp. 199–245, 2005.

[28] D. Jackson, "Aspect: detecting bugs with abstract dependences," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 109–145, 1995.

[29] D. Evans, "Static detection of dynamic memory errors." in *Proceedings of the Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 1996, pp. 44–53.

[30] D. L. Detlefs, "An overview of the extended static checking system," in *Proceedings of the First Formal Methods in Software Practice Workshop (1996)*, 1996.

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

[32] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[33] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved, "Saber: smart analysis based error reduction," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM Press, 2004, pp. 243–251.

[34] Analyst4j, February 2008, http://www.codeswat.com/.

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

[36] CheckStyle, 2004, http://checkstyle.sourceforge.net.

[37] FXCop, June 2006, http://www.binarycoder.net/fxcop/index.html.

[38] Hammurapi, October 2007, http://www.hammurapi.biz/.

[39] SemmleCode, October 2007, http://semmle.com/.

[40] D. Beyer, A. Noack, and C. Lewerentz, "Efficient relational calculation for software analysis," *Transactions on Software Engineering*, vol. 31, no. 2, pp. 137–149, February 2005.

[41] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to software engineering." *Int. Journal on Software Tools for Technology Transfer*, vol. 9, pp. 505–525, 2007, invited to special issue of selected papers from FASE 2005.

[42] H. Chen and D. Wagner, "Mops: an infrastructure for examining security properties of software." in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002, pp. 235–244.

[43] R. Prieto-Díaz, "Domain analysis: An introduction," *Software Engineering Notes*, vol. 15, no. 2, pp. 47–54, April 1990.

[44] R. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley Professional, 2002.

[45] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Proceedings of the* 39^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, Q. Li, R. Riehle, G. Pour, and B. Meyer, Eds. IEEE Computer Society Press, July 2001, pp. 296–305.

[46] S. Boroday, A. Petrenko, J. Singh, and H. Hallal, "Dynamic analysis of Java applications for multithreaded antipatterns," in *Proceedings of the* 3^{rd} *International Workshop On Dynamic Analysis*. New York, NY, USA: ACM Press, 2005, pp. 1–7.

[47] B. Dudney, S. Asbury, J. Krozak, and K. Wittkopf, *J2EE AntiPatterns*. Wiley, 2003.

[48] B. A. Tate and B. R. Flowers, *Bitter Java*. Manning Publications, 2002.

[49] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA, USA: Addison-Wesley Professional, 2002.

[50] Janice Ka-Yee Ng and Y.-G. Guéhéneuc, "Identification of behavioral and creational design patterns through dynamic analysis," in *Proceedings of the* 3^{rd} *International Workshop on Program Comprehension through Dynamic Analysis*, A. Zaidman, A. Hamou-Lhadj, and O. Greevy, Eds. Delft University of Technology, October 2007, pp. 34–42, tUD-SERG-2007-022.

[51] J. M. Chambers, W. S. Clevelmd, B. Kleiner, and P. A. Tukey, *Graphical methods for data analysis*. Wadsworth International, 1983.

[52] R. Marinescu, "Measurement and quality in object-oriented design," Ph.D. dissertation, Politehnica University of Timisoara, June 2002.

[53] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[54] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the UML cake," in *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, D. C. Schmidt, Ed. ACM Press, October 2004, pp. 301–314.

[55] C. Consel and R. Marlet, "Architecturing software using: A methodology for language development," *Lecture Notes in Computer Science*, vol. 1490, pp. 170–194, September 1998.

[56] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Proceedings of the* 26^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, J. Gil, Ed. IEEE Computer Society Press, August 1998, pp. 112–124.

[57] DECOR, September 2006, http://ptidej.dyndns.org/research/ptidej/decor/.

[58] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*, 1^{st} ed. MIT Press, July 1991.

[59] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, December 2005.

[60] Y.-G. Guéhéneuc, H. Sahraoui, and Farouk Zaidi, "Fingerprinting design patterns," in *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, E. Stroulia and A. de Lucia, Eds. IEEE Computer Society Press, November 2004, pp. 172–181.

[61] H. Albin-Amiot, P. Cointe, and Y.-G. Guéhéneuc, "Un méta-modèle pour coupler application et détection des design patterns," in *Actes du* 8^{e} *colloque Langages et Modèles à Objets*, ser. RSTI – L'objet, M. Dao and M. Huchard, Eds., vol. 8, numéro 1-2/2002. Hermès Science Publications, janvier 2002, pp. 41–58.

[62] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 – the FAMOOS information exchange model," University of Bern, Tech. Rep., 2001.

[63] A. Winter, B. Kullbach, and V. Riediger, "An overview of the gxl graph exchange language." in *Software Visualization*, ser. Lecture Notes in Computer Science, S. Diehl, Ed., vol. 2269. Springer, 2002, pp. 324–336.

[64] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 262–292, 1996.

[65] H. A. Muller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *ICSE — Future of SE Track*, 2000, pp. 47–60.

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

Naouel Moha received the Master degree in computer science from the University of Joseph Fourier, Grenoble, in 2002. She also received the Ph.D. degree, in 2008, from the University of Montreal (under Professor Yann-Gaël Guéhéneuc's supervision) and the University of Lille (under the supervision of Professor Laurence Duchien and Anne-Françoise Le Meur). The primary focus of her Ph.D. thesis was to define an approach that allows the automatic detection and correction of design smells, which are poor design choices, in object-oriented programs. She is currently a postdoctoral researcher in the INRIA team-project Triskell. Her research interests include software quality and evolution, in particular refactoring and the identification of patterns.

Yann-Gaël Guéhéneuc is associate professor at the Department of computing and software engineering of Ecole Polytechnique of Montreal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals.

Laurence Duchien obtained her Ph.D degree from University Paris 6 LIP6 laboratory in 1988 and she worked on protocols for distributed applications. She joined then the Computer Science Department at CNAM (Conservatoire National des Arts et métiers) (http://www.cnam.fr), Paris, France as associate professor in September 1990. She also holds a Research Direction Habilitation in Computer Science from the University of Joseph Fourier, Grenoble, France in 1999. She is currently full professor at the computer science Dept at University of Lille, France since 2001 and she is the head of the INRIA-USTL-CNRS team-project ADAM (Adaptive Distributed Applications and Middleware) (http://adam.lille.inria.fr). Her current research interests include development techniques for component-based and service-oriented distributed applications in ambient computing. She works on the different steps of life cycle development such as architecture modeling, model composition and transformation and, finally, software evolution.

Anne-Françoise Le Meur received her Master of Science in Computer Science at the Oregon Graduate Institute in Portland Oregon USA in 1999 and her PhD in Computer Science from the University of Rennes 1 in 2002. After a one year postdoc at DIKU University of Copenhaguen in Denmark, she obtained in 2004 an associate professor position at the University of Lille 1 and joined the INRIA team project ADAM. She has worked on program specialization, and the design and development of domain-specific languages. Her current work focuses mainly on the application of programming-language techniques to the problem of software component-based architecture conception and evolution.