Computer Science Department
University of Valladolid
Valladolid - Spain

A Classification Framework and Survey for Design Smell Management

Javier Pérez, Carlos López, Naouel Moha, Tom Mens

Departamento de Informática, Universidad de Valladolid, Spain.
jperez@infor.uva.es
Departamento de Ingeniería Civil, Universidad de Burgos, Spain.
clopezno@ubu.es
Département d'informatique, Université du Québec à Montréal, Canada.
moha.naouel@uqam.ca
Service de Génie Logiciel, Université de Mons - UMONS, Belgium.
tom.mens@umons.ac.be

Abstract. Many approaches and tools have been developed to detect, correct or reduce design smells. This technical report presents a taxonomy for design smell management approaches, using feature diagrams as a graphical guidance to illustrate it. This taxonomy can help in various ways. Newcomers in the domain can use it to get acquainted with the important aspects of design smell management, tool builders may use it to compare and improve their tools, and software developers may use it to assess which tool or technique is most appropriate to their needs.

Technical Report No. IT-DI-2011-0001

Table of Contents

List of Figures

1 INTRODUCTION

Software evolution is a fundamental activity of software development that often results in an increase of software entropy and, as a consequence, in the decay of software structure. As a software system evolves, its structure tends to deteriorate because the evolution efforts concentrate more on the correction of bugs and on the addition of new functionality than on the control and improvement of the software's architecture and design [Bro75].

Design smells are problems encountered in the design of a software system, and stem from "poor" design choices, leading to ill-structured software. This may hinder further development and evolution by making it harder for software developers to change the software. Design smells can arise at different levels of granularity, ranging from high level design problems, such as antipatterns [BMMIM98], to low-level or local design problems, such as code smells [FBB+99].

Bad design practices, often due to inexperience, insufficient knowledge or time pressure, are at the origin of design smells [PW92]. Often, these are not isolated problems, but symptoms or precursors of more global defects. Design smells are quite different from software *defects* (also referred to as *bugs*) which are "deviations from specifications or expectations which might lead to failures in operation" [FN99,Hal77].

An increasing number of approaches have been proposed for both the *detection* and *correction* of design smells. The proposed *detection* techniques consist mainly in defining and applying rules for identifying design smells. As for the *correction* techniques, they often consist in suggesting which *refactorings* could be applied to the source code of a system to restructure it, thereby correcting, or at least reducing, its code and design problems. In this technical report, we refer to **design smell management** as the collection of techniques, tools and approaches to detect, correct or reduce design smells. Correcting, or at least, reducing smells can improve software quality. As such, the goal of a smell management process is to change the system's structure to improve its internal quality factors, in particular its understandability and maintainability[1].

This technical report presents the results of a literature study that we have carried out on design smell management techniques. We have also downloaded and evaluated numerous tools that automate some of the design smell management activities. The results of our taxonomy are presented as a concise multi-dimensional classification framework, using feature diagrams as a graphical representation [CE00]. Our framework can be used, among others, to compare the commonalities and variabilities of different approaches and tools based on particular criteria of interest.

Due to the abundance of research literature available on the subject, we have deliberately restricted our study in various ways. First of all, we only consider approaches that are directly related to the management of structural problems

[1] A good reference on software quality factors can be found in the ISO 9126 standard [ISO01].

that can be detected in software code and design, in particular for object-oriented software. A lot of related work exists on the identification and correction of different kinds of problems in specific types of software-related systems, such as databases [BGQR07,JFRS07], and networks [PP07], but these are outside the scope of this technical report. The specification and detection of design smells relate more generally to the field of design pattern specification and detection (*e.g.,* [GA08]). Such approaches are also outside the scope of our technical report. Finally, two major approaches to design smell management can be distinguished: preventing smells before they occur; and correcting them once they have appeared. These two approaches are so different by nature that a taxonomy covering both of them would not be very meaningful. The taxonomy presented in this technical report therefore focuses on the second approach only.

The remainder of the technical report is structured as follows. Section 2 introduces the necessary terminology and historical background on design smells. Section 3 explains how our taxonomy is structured and illustrated, using the feature diagram notation. The next three sections each focus on a major variation point enabling to distinguish smell management approaches: the design smells addressed (Section 4) , the targeted artefact(s) (Section 5), and the supported activities (Section 6). Finally, Section 7 concludes the technical report, by discussing open problems and by presenting challenging avenues of future research.

2 DESIGN SMELLS DEFINITIONS AND HISTORICAL BACKGROUND

First of all, a precise definition of the term "smell" is needed. A "smell" describes a situation suggesting a potential problem. To decide whether the problem is real and relevant, the situation has to be examined in more detail. Since the introduction of the term "smell", there has been a plethora of different terms to refer to a family of very similar concepts. This section aims to clarify the terminology and proposes a way to unify the different terms appearing in the "smell" literature.

The term "defect" has also been used to refer to these kind of design problems in some articles [GAA01,MGLM+09], but the term is more often used to refer to other problems. Travassos et al. [TSFB99], for example, use the term "defect" to describe quality problems caused mainly by inconsistencies between system requirements and system design. In [RSS+04] the authors present an approach to detect software errors which are also referred as "defects". These errors include resource management bugs, concurrency problems, server side poor performance, persistent data management bugs and implementation contract violation. This is how "defect" is understood, in general, standing for "software defects" or software "bugs" [RSG08]. According to Fenton et al. [FN99] "software defects" are *"deviations from specifications or expectations which might lead to failures in operation"*. In this technical report we do not pretend to cover approaches related to this kind of problems. The term "flaw" has also been used to refer to

"smells" [CLMM06,LWN07a,Mar02,Tri08]. However, it is more often associated with run-time and compile-time errors [EGR+06].

For all of the above reasons, we consistently use the term "smell" to distinguish it from "defect" or "flaw". Moreover, as stated previously, we only focus on smells related to object-oriented software. We present in the following two categories of smells: code and design smells.

2.1 Code Smells

The term "code smell" was introduced by Kent Beck[2] to define structural problems in source code that can be detected by experienced developers:

> "A code smell is a hint that something has gone wrong somewhere in your code."

The suspect structure may not be causing serious harm (in terms of bugs and failures) at the moment, but it has a negative impact on the overall structure of the system and, as a consequence, on its quality factors. Code smells can clutter the design of a system, making it harder to understand and maintain. Moreover, the presence of code smells can warn about wider development problems such as wrong architectural choices or even bad management practices. The term became popular with the book from Fowler *et al.* [FBB+99], where a compilation of "bad smells in code" can be found. A brief description of the term can also be extracted from the book:

> "... structures in the code that suggest (or sometimes scream for) the possibility of refactoring."

This description states the relationship between code smells and refactorings. Code smells reveal where and how to refactor, and inversely, refactorings become the preferred way to remove code smells.

A well-known example of code smell is the presence of duplicated code. This situation appears when a concept, an expression or an algorithm is used in many places across a software system. Because it has not been correctly identified within the design, it has been copied wherever needed. Duplicated code makes the system's structure unnecessarily more difficult to maintain because it becomes a source of multiple problems. Errors in the duplicated code can spread over the system, changes to be made to a duplicated portion of code should also be applied to the copies, etc. The code should therefore be refactored in order to unify the duplicated structures into a single definition of the concept it represents. Duplicated code can be the consequence of a design error, a quick patch to achieve a tight deadline, or a deep-rooted bad practice of copy-paste reuse. Various tools and techniques have been proposed to detect duplicate code, such as DupLoc [DRD99] and its descendants, or the Copy-Paste Detection (CPD) part of the PMD tool [PMD09].

[2] See http://c2.com/cgi/wiki?CodeSmell

2.2 Design Smells

We refer to "design smells" as a concept that includes "code smells", while being more general. The term covers the whole range of problems related to the software's design structure.

Different smells affect the software at a different granularity level, from methods (*e.g.,* Long Parameter List [FBB+99, page 78]) to the whole system architecture (*e.g.,* Stovepipe System [BMMIM98, page 159]). Due to this, different authors have made further additions to the smell terminology. The design smell concept appears in the literature with a variety of terms, for example design flaws in [SLT06,Tri08] disharmonies in [LM06], or defects in [TMMM07]. In [MGDLM10], a distinction is made between smells appearing in the refactoring book from Fowler *et al.* [FBB+99], and in the antipatterns book from Brown *et al.* [BMMIM98], being identified as code smells and design smells, respectively. Martin [Mar03] refers to "design smells" as higher-level smells that cause the decay of the software system's structure, which can be detected when software starts to exhibit the following problems:

- **Rigidity**: The design is hard to change because every change forces many other changes to other parts of the system.
- **Fragility**: The design is easy to break. Changes cause the system to break in places that have no conceptual relationship to the part that was changed.
- **Immobility**: It is hard to disentangle the system into components that can be reused in other systems.
- **Viscosity**: Doing things right is harder than doing things wrong. It is hard to do the right thing because sometimes it's just easier to do "quick hacks".
- **Needless Complexity**: The system is over-designed, containing infrastructure that adds no direct benefit.
- **Needless Repetition**: The design contains repeating structures that could be unified under a single abstraction.
- **Opacity**: The system is hard to read and understand and does not express its intent well.

In spite of the different terminology used in the literature, all aforementioned terms have in common that they describe problems related to a bad design, such as a misidentified or overlooked abstraction, a misused pattern, an under- or over-engineered design, etc. For the sake of simplicity, and in order to unify the existing terminology, we use the term "design smell" to represent:

> a problem encountered in the software's structure (code or design), that does not produce compile-time or run-time errors, but negatively affects software quality factors. This negative effect could lead to errors in the future.

2.3 Historical Background on Design Smells

Several books relate to design smells. Webster [Web95] wrote the first book on smells in the context of object-oriented programming, including conceptual,

political, coding, and quality-assurance pitfalls. Riel [Rie96] defined 61 design heuristics characterising good object-oriented programming, enabling software engineers to assess the quality of their systems manually, and providing a basis for improving design and implementation.

Beck [FBB+99] compiled 22 code smells, which suggest where and when the engineers should apply refactorings. Code smells are described in an informal style and associated with a process to locate them through manual inspections of the source code.

Brown *et al.* [BMMIM98] provided "antipatterns" as another type of smell specification. They focused on the whole software development process of object-oriented systems and textually described 41 antipatterns, which are general object-oriented smells. Fourteen of these antipatterns relate to software development (design smells), 13 of them relate to software architecture, and the remaining 14 relate to project management. Belonging to the software development category are the well-known antipatterns Blob and Spaghetti Code. In this technical report, we are not interested in managerial antipatterns since they are quite difficult to detect in design workproducts.

Tiberghien *et al.* [TMMM07] described 47 design smells, most of which taken from the other smell catalogues. They use the French term "défauts de conception". The smells listed are all about bad design, except for "code level defects", which address low-level code problems not so much related to design.

These books and catalogues provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide audience for educational purposes. However, manual inspection of the code for searching smells based only on textual descriptions is a time-consuming and error-prone activity. Some works have analysed and compiled design smells with more detailed and structured descriptions to ease their manual or automated detection and correction.

Kerievsky [Ker04] added some new smells to Beck's catalogue, such as: Solution Sprawl, Oddball Solution, Indecent Exposure and Combinatorial Explosion. In addition, he related code smells to refactorings and design patterns. Some smells can originate from the misuse or the absence of design patterns, and therefore these smells can be corrected by introducing or removing design patterns. For example, *Move Creating Knowledge to Factory* is a refactoring that solves the smell of sprawling object creation responsibility by introducing a factory design pattern.

Lanza and Marinescu [LM06] presented a catalogue of design smells called "disharmonies", along with the definition of *detection strategies* and recommendations for the correction process. This approach introduces metrics-based rules to capture deviations from "good" design principles. The term "disharmony" can be understood more as an intent to unify the terminology than as a kind of defect on its own. Many disharmonies are similar to code smells in terms of their abstraction level. Indeed, 7 out of 11 disharmonies are problems already referenced in [FBB+99]. The definition of "detection strategies" is a step towards precise specifications of design smells that can allow the automated smell detec-

tion. The recommendations for correction are not so suitable to automation, but are sufficiently structured to guide the developer through the correction process.

Trifu [Tri08] described a set of 10 design smells, which he coined "design flaws", and provided "restructuring patterns" to detect and correct them. Restructuring patterns identify not only the smell, but also the intent of the design. This is then used to propose a correction strategy, which is a pseudo-code algorithm based on the application of refactorings and on the introduction of design patterns.

Moha *et al.* [MGDLM10] proposed a method and a technique to specify, detect and visualise design smells. They used a domain-specific language to allow the user to specify smells. These specifications are transformed into generated Java source code, which can be compiled and run to search for smells. The specification process is manual but the detection and visualisation are fully automated through DECOR, the tool that implements the approach.

From the above, it should be clear that the research in the field of design smells has evolved to propose more automated approaches, not only to smell detection, but also to the other activities of smell management, such as specification and correction.

3 OVERVIEW OF THE TAXONOMY

The goal of this technical report is to present a *taxonomy* of design smell management approaches, providing a framework to compare and analyse the current and future design smell management techniques and tools. This taxonomy can be used for a wide variety of purposes. Among others, it can help software developers choosing a particular approach that is best suited for their needs, it can help tool builders to assess the strengths and weaknesses of their tool compared to other tools, and it can help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

3.1 Feature modelling notation

As a visual aid to guide our design smell management taxonomy, we rely on a visual notation called feature diagrams, that is inspired by the one used by Czarnecki and Helsen to present their survey on model transformation [CH06]. *Feature diagrams* are a visual representation of feature models. Different symbols and notations for feature diagrams can be found in the literature. The notation used in this technical report is shown in Figure 1.

Feature modelling [CE00] is the activity of modelling the common and variable properties of concepts and their interdependencies by organising them into a coherent model referred to as a *feature model*. This model is used to represent a hierarchy of features, representing the common and variable properties of concept instances and the dependencies between the variable features.

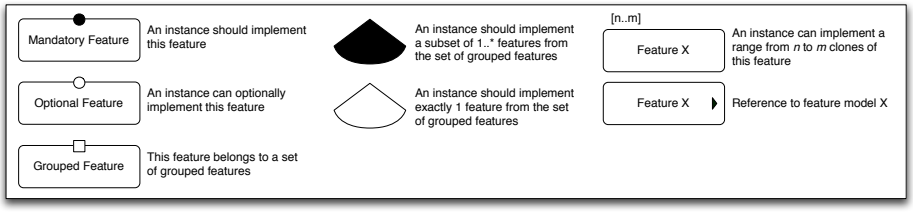

Fig. 1. Feature diagram notation used throughout this survey.

Feature models are a nice and intuitive way to represent a family of systems, or a concept such as design smell management, through the analysis of the commonalities and differences between the wide variety of approaches supporting it. Features are important properties of a concept, and the aim of feature modelling is to describe a family or a concept, within a given domain, through the analysis and specification of its particular occurrences. Features also serve to capture and model the knowledge and terminology of that domain. The basis for feature modelling can be found in Czarnecki and Eisenecker's book [CE00].

During the analysis of the surveyed design smell management approaches and tools, we have detected many commonalities shared between all of them, or between some subsets. For example, all approaches address a certain type of target artefact. In the same way we have identified relevant differences that can be used to characterise each approach. As an example (see Section 5), an approach can search for design smells in source code, or in executable code, models, etc. Consequently, for this taxonomy, we have found feature diagrams to be an appropriate and useful notation. Moreover, since all design smell management tools belong to a "family" of approaches, they can be naturally described with feature diagrams.

The reader should note that this survey, and the feature diagrams that illustrate it, reflect the current state of the art. Therefore, the features identified in some categories, such as "Smell Property" in Section 4 or "Type of Representation" in Section 5, may be extended by future approaches. For this reason, we have sometimes added to the feature diagrams elements that do not match current approaches, but rather illustrate what we consider to be feasible or desirable in future approaches. Such is the case of the "Target Artefact" multiplicity in Figure 2.

3.2 Top level features of the design smell management taxonomy

The proposed feature diagram notation allows us to group design smell management activities, tools, techniques or formalisms based on their commonalities. We adopt a multi-dimensional classification, allowing us to describe and compare different approaches, based on the criteria of interest. Within our classification, we do not consider general properties such as interoperability, usability, or extensibility because these are tool-specific properties that can apply to any kind

of tools regardless of the domain of interest. As such, they are not specific or intrinsic to design smell management approaches *per se*.

To present our survey in a structured way, each of the following sections discusses the main features with respect to design smell management that can be used to group together approaches sharing these features. We start by describing the top level features, constituting the main common properties of our field of study. We then descend down the model to describe each of the subfeatures. The root feature of our model is *Design Smell Management*. It represents any approach dealing with design smell management. An instance of the model will represent an incarnation of an existing approach, or even a non-existing one that would be feasible and interesting to develop. The three top-level features of the *Design Smell Management* root feature are shown in Figure 2 and explained below. These features describe properties common and mandatory to every design smell management approach.

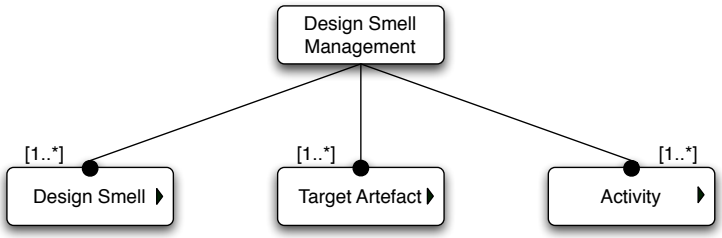

Fig. 2. Top-level variation points for comparing design smell management approaches.

Design Smell. A wide range of design smells can be managed by different approaches. The nature of the smells each approach addresses is a major top-level variation point.

Target Artefact. Any design smell management tool requires at least one software artefact on which the smell can be observed. We will refer to this (set of) software artefact(s) as the *Target Artefact* of the approach.

Activity. The third top-level variation point for design smell management approaches is the set of activities explicitly supported by each approach. An example activity is smell specification. Every approach requires a definition of design smells in order to be able to detect them, but only some approaches present explicit support for the smell specification activity.

The next three sections of this technical report will discuss each of these three main features in detail, by further decomposing them into subfeatures and using them to survey the state of the art in design smell management.

4 DESIGN SMELL

As introduced by the root feature diagram in Figure 2, a design smell management approach can cover several smells. The *Design Smell* feature, depicted in Figure 3, allows to describe in more detail the nature of those smells. This feature is split into five branches of mandatory features. Most approaches are specialised in: (1) a certain type of smells, (2) smells at different levels of abstraction, (3) smells affecting several program entities at a particular level of granularity, and (4) smells that have internal properties of different natures. This classification of design smells extends those proposed in [MLC06,MGDLM10]. Although we use it for the purpose of establishing a classification framework for design smell management approaches, it is also useful on its own, for classifying design smells themselves.

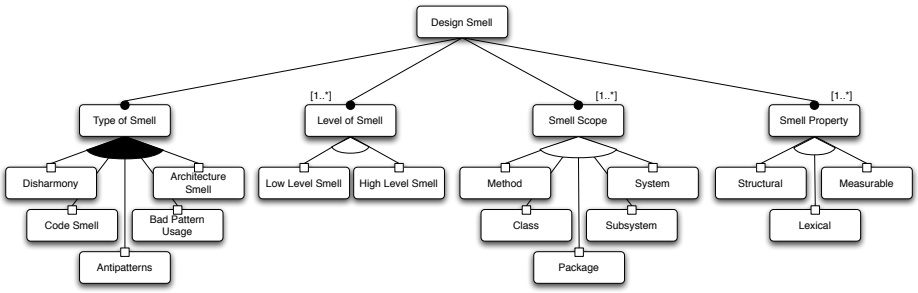

Fig. 3. Top-level *Design Smell* feature, and its subfeatures.

Type of Smell. This feature describes the type of design smells addressed by an approach. By "type of smell" we refer to the catalogue in which the smell is defined. Some design smell management approaches define their own set of smells [LM06], but we the vast majority of them are focused in those smells described in a small number of catalogues. Those represented in the feature diagram of Figure 3 are the most widely referenced catalogues. To characterise a design smell management approach, we should describe which catalogue(s) of smells it addresses.

Marinescu *et al.* [LM06] define *disharmonies* as design smells that affect single entities such as classes and methods. The particularity of these disharmonies is that their negative effect on the quality of design elements can be noticed by considering these design elements in isolation. They identify three aspects that contribute to identify disharmony of a single entity: its size, its interface and its implementation. iPlasma [Gro], InCode and InFusion [Int08] are three tools that detect these disharmonies, using object-oriented metrics with customized filters.

Most of the studied approaches focus on code smells specified in the catalogue of Fowler [FBB+99], such as [AS09,CLMM06,TCC08,Mun05,Sem07]. Munro proposed metric-based heuristics with thresholds to detect code smells [Mun05], which are similar to Marinescu's detection strategies [Mar02]. Alikacem and Sahraoui [AS09] proposed a language to detect smells and violations of quality principles in object-oriented systems. This language allows the specification of rules using metrics, inheritance, or association relationships among classes, according to the user's expectations. It also uses fuzzy logic to express the thresholds of rule conditions.

The DECOR method developed by Moha *et al.* [MGDLM10] focuses mainly on Brown's *antipatterns* [BMMIM98]. As another example, Analyst4j [Ana08] allows the identification of antipatterns and code smells in Java systems using metrics.

Some approaches use *design patterns* to search for design smells [GHJV95], either to look for opportunities to apply a design pattern or to detect misapplication of patterns [GAA01,Ker04,TM03]. Guéhéneuc *et al.* [GAA01] use design patterns as reference structures, and detect design smells as failed intents of applying design patterns. For this, they search for structures that ressemble design patterns but slightly deviate from them. To correct these smells, they suggest to transform these structures so they match the intended design pattern properly. In [Ker04], Kerievsky proposes a more manual approach for detection and correction of design-pattern-related smells. He instructs the developer to search for structures that reveal that a design pattern has been misapplied, but he also describes situations where a design pattern is absent and can be introduced, and situations where a design pattern is cluttering the design and therefore, should be completely removed.

Roock and Lippert [SR06] present a catalogue of 'architecture smells" related to the organization of subsystems. Among these smells we can find *dependency-related problems* such as cyclic dependencies. Such smells can be detected by tools like [Con99,STA,Tes08].

Level of Smell. Another way to classify design smells is through the distinction between low-level and high-level smells. *Low-level smells* focus on a single very specific problem. Code smells [FBB+99] such as are "long methods", "data classes", and "large classes" are examples of low-level smells that refer to very specific situations observed in the code. *High-level smells* are design smells that are composed of other smells, such as antipatterns. They focus on a variety of similar (but different) problems. An example of a high-level smell is the Blob antipattern [BMMIM98], also known as God Class. It reveals a procedural design (and thinking) implemented with an object-oriented programming language. It manifests itself through a large controller class that plays a God-like role in the program by monopolizing the computation, and which is surrounded by a number of smaller data classes providing many attributes but few or no methods. This high-level smell is composed of other low-level smells such as the code smells "data class" and "large class".

Smell Scope. This feature is used to describe the extent or scope of the different types of entities involved in the supported smells. For most object-oriented languages, the different scopes would be system, subsystem, package, class, method and statement. High-level smells usually represent design problems with a wide scope, affecting several and/or large entities. Low-level smells, on the contrary, have an effect over a well-defined and limited scope, *i.e.,* within a single and small entity. As illustrated by the multiplicity associated to this feature, a design smell can extend over several entities belonging to different levels.

Smell Properties. The nature of a code smell can be summarised and represented by the smell indicators or*properties* used in its specification. These properties can be decomposed into: *structural* descriptions of "smelly" structures in the design of code; *measurable* specifications based on metrics and measurable properties of the system; and *lexical* definitions based on the names of the software entities. Design smell management approaches benefit from those properties in order to tackle a particular activity. For example, in the Blob description, the large class corresponds to a measurable property that can be easily computed by counting the number of methods and attributes, whereas the data class is a structural property consisting of identifying accessor methods. A lexical property in the Blob corresponds to the use of procedural names (such as Main, Make, Create, Exec) used in the classes affected by such smell. iPlasma [Gro,LM06] uses the measurable properties of a Blob – referred to as God Class – to detect it, while DECOR [MGDLM10,Tea09] additionally uses the lexical property to perform the detection.

5 TARGET ARTEFACT

The *Target Artefact* feature is a major variation point to distinguish design smell management approaches. This feature refers to the software artefacts on which the smells can be observed. It is shown in Figure 4, and its subfeatures are presented below.

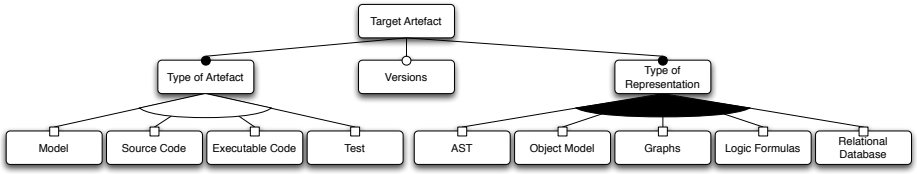

Fig. 4. Top level *Target Artefact* feature, and its subfeatures.

5.1 Type of Artefact

Any approach addressing the management of design smells, should focus on, at least, one type of software artefact. The types of artefacts supported by an approach, and the way they are represented internally, are tightly coupled to which smells can be managed and how.

A particular tool or technique often targets only a single type of artefact. In fact, we could not find a tool that supports many types of artefacts. Nevertheless, it is feasible and desirable to build a tool that uses different types of artefacts as complementary sources of information, thereby improving its results. The manual detection process described by Travassos *et al.* [TSFB99] illustrates the feasibility of this. To identify code smells, the developer is instructed to examine different types of models, such as requirements descriptions, use cases, class diagrams, class descriptions, state diagrams, and interaction diagrams. This is shown in Figure 4 as a decomposition of feature *Type of Artefact* into a set of OR grouped subfeatures. The most common types of artefacts are source code and bytecode or executable code. In addition, several approaches support detection of design smells by analysing software models. A tendency of modelling tools is, for example, to provide model warnings, so-called "critics", to the user. This is done by ArgoUML [Arg] and Together [Bor].

The vast majority of available tools aim at managing source code smells. CheckStyle [Che04] and Hammurapi [Ham07], for example, load Java code and search for violations of coding standards. The concept of design smell is language-agnostic, so many other programming languages may be supported as well. For example, Reek [Rut] and Roodi [Roo] search for design problems in Ruby source code. Some authors, such as Ciupke [Ciu99] and Sahraoui *et al.* [SGM00] analyse smells in C++. FXCop [FXC06] and StyleCop [Sty] find deviations from code conventions in C# code. Some tools provide multi-language support, such as Decor [MGDLM10], iPlasma [Gro,LM06] and the newest iPlasma versions InCode-InFusion [Int08], which supports the analysis of C++, Java and C# programs.

Another widely supported type of artefact is executable code –*e.g.,* binary code or bytecode–. Tools such as RevJava [Rev] and Stan4J [STA]. analyse Java bytecode. The advantage of addressing executable code is that these tools can be used even when the source code is not present.

An emerging trend in many software fields, and especially in *agile software development*, is to treat *tests*, and more precisely scripted tests [Mes07], as first-class citizens. An increasing number of authors have addressed the problem of smells in scripted tests [Mes07,vMvK01] and propose approaches that provide support for managing design smells in scripted tests [NB07,VRDBDR07].

5.2 Versions

A design smell management approach can benefit from the additional information that can be extracted from a version repository that stores multiple versions

of the target artefact(s) under study. Some code smells from [FBB+99], such as *Shotgun Surgery*[3] are more easily identified by analysing the change history of the system. Support for multiple versions of an artefact is an optional subfeature that it is applicable to any type of artefact.

Several approaches [GDMR04,RSG08,XS04] propose to use different versions of the target artefact as input. Their supporting tools often include support to access version repositories (*e.g.,* CVS, SVN, GIT, etc.), and to extract and analyse the software artefacts and their metadata from these repositories. Some of these approaches [GFGP06,LWN07a] even claim that this is the only way to detect some particular smells or to obtain a wider picture of a certain problem.

5.3 Type of Representation

Every design smell management approach is based on an internal representation of the software artefact the approach is dealing with. In order to analyse and process the targeted artefact, its internal representation will be used. This feature is relevant because there is a strong dependency between the internal representation and other aspects such as the technique, the expected results or the automation support.

A common way of representing a software artefact is by means of an *Abstract Syntax Tree* (AST). This type of representation is especially frequent in approaches that target source code or executable code. A typical AST representation will keep the complete information available in the examined artefact. Different approaches [TCC08,Sli05,TK04,TCC08] can simplify the AST to keep just the relevant information needed for the task at hand, or even augment the AST with additional details in order to ease the design smell management activities.

Other tools rely on a representation based on some *Object Model*. They use a specific metamodel of the targeted artefact, such as MOON [Cre00,CLMM06], FAMIX [Tic01,LM06] or MEMORIA [Raţ04,Gro] This type of representation is mostly used to simplify the target artefact. Just the information needed by the approach is extracted from the targeted artefact. Analysis and manipulation of the targeted artefact is performed by programmed procedures using a wide variety of programming languages, even custom-built Domain-Specific Languages (DSL) [Gué03,MGLM+09].

Graph-based approaches can also be employed to analyse the target artefact in search of design smells [vM02]. Graph theory can help analyse artefacts in search of defects, and graph transformation techniques to apply corrections [MVDJ05]. De Lucia *et al.* [DLOV08] represent classes as graphs and use measurable properties of these graphs to find refactoring opportunities that will improve the cohesion of that classes.

Logic Formulas offer a similar formal support to represent software. This type of representation enables the use of logic-based techniques to manage design smells [Ciu99,TM03].

[3] Whenever a change has to be made to a part of the system, many more little changes to other parts of the system are needed too.

Some tools store the information extracted from the targeted artefact in a *Relational Database*. This usually speeds up the task of querying the software model by taking advantage of a dedicated and specialised query engine. This type of representation can be used just internally by a tool [GFGP06,SSL01,TSG04], or even be offered to its final users, enabling them to analyse the artefact by means of structured and composable queries. For example, with SemmleCode [Sem07], engineers can execute queries against source code, using a declarative query language called .QL, to detect code smells.

These different types of representation can be combined. An artefact represented as an AST can be analysed with graph algorithms, if it is formalised with graphs, or logic formulas. A model can be stored in a relational database in order to provide an easy and efficient way to access it by querying the database.

6 ACTIVITY

The *Activity* feature, depicted in Figure 5, represents a major variation point among design smell management approaches. The design smell management process can be decomposed into different types of activities that can be supported by a particular approach. We have decomposed this feature into 5 types of activities that we have found in all approaches we have surveyed. For each type of activity we describe the *techniques* used to support it, the *automation support* achieved and the type of *results* that the different approaches produce.

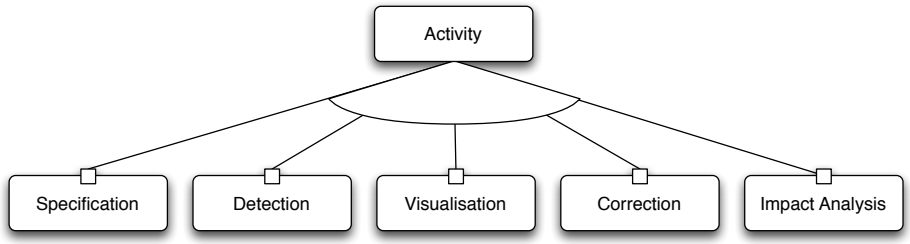

Fig. 5. Top level *Activity* feature, and its subfeatures.

The *automation support* of an activity reflects the maturity of the studied approach in supporting this activity. We distinguish between the following levels of automation, depicted in Figure 6, which simplify the ones defined by Sheridan [She00]:

Manual: The activity is carried out in a manual way.
Suggest Alternatives: The tool can execute the activity automatically and suggest options or alternatives to the user. The user still needs to select and apply the suggestion manually.

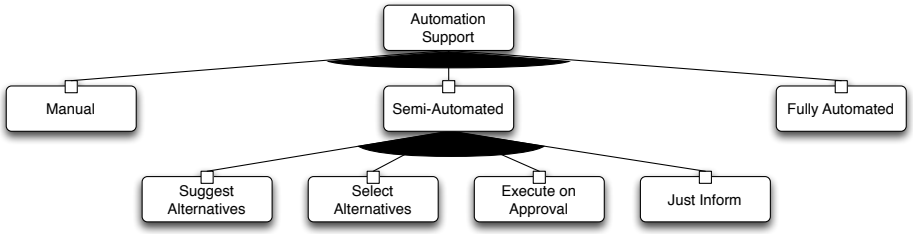

Fig. 6. Degree of automation support of a design smell management activity.

Select Alternatives: The tool suggests and selects the alternative tasks to be performed. The user needs to confirm this selection.

Execute on Approval: The tool presents the user the activity that is going to be executed, but requests permission. The user can only choose to apply the activity as a whole, or to cancel it.

Just Inform: The tool decides and executes the activity without asking the user, but informs the user about the process.

Fully Automated: The tool performs the activity in a fully automatic way, without informing the user of what is happening.

Any design smell management approach must produce some results when applied to a software artefact. To compare approaches, we found it necessary to describe the type of *result* that is being, or can be, obtained from each activity. Analysing an approach or a tool according to this dimension is important in order to determine whether it is useful to solve a particular problem, or whether it can be combined with another approach or tool.

In each of the following subsections, we survey the current state-of-the-art of supporting a particular design smell management activity, in the level of detail that we have explained above.

6.1 Specification

The *specification* activity is implemented by those approaches that provide the developers the necessary support to extend or adapt it to their particular needs, by specifying new design smells or modifying existing smells.

Technique. Most of the surveyed approaches that support the activity of specification include a description, at least textual, of the design smell for detecting or correcting it. Typically, the technique used by tools to specify design smells is through the use of (possibly customisable) rules expressed in some *formal language* (e.g. OCL, SQL, XPath, logic formulas), *programming language* (e.g. Java), or *domain-specific language* ([MGLM+09]). The design smell specifications in [FBB+99], include a description of general guidelines to correct them. Wake [Wak03] specifies these correction guidelines in the form of recipes.

The antipatterns book [BMMIM98] provides a specification in terms of counter-examples.

Automation. Several approaches or tools allow writing user-defined detection or correction rules such as PMD [PMD09], Eclipse's Metrics plugin [Ecl], RefactorIt [Aqr02], iPlasma [Gro,LM06], Decor [MGDLM10] and CoderAider [Dur07]. However, this activity is intrinsically manual. The ability to define or tune the smell detection rules is common to most approaches that deal with smells based on metric warnings. SemmleCode [Sem07] provides a quite versatile interface to specify design smells, through its built-in query language that can be used to write complex queries to detect smells.

Result. The *specification* activity produces either a purely descriptive and non automatable *design smell specification*, or some kind of *design smell detection rules* that can be automated or even consists of an executable *detection program*.

Figure 7 summarises the techniques, automation support and results for the specification activity.

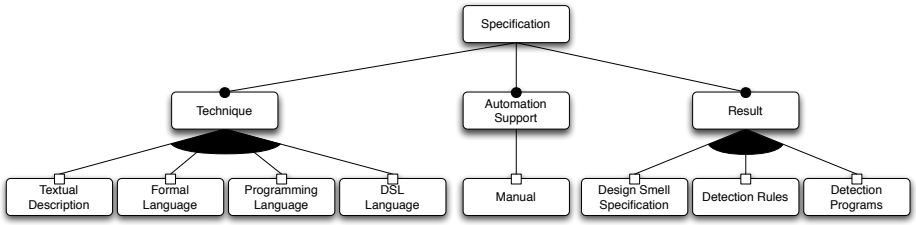

Fig. 7. Summary of the current situation for the specification activity.

6.2 Detection

The vast majority of existing design smell management approaches focus on the activity of *design smell detection*.

Technique. One of the approaches to detect design smells is through *manual code inspection* [TSFB99]. Most detection approaches, though, are based on the use of *metrics*. The vast majority of metric-based approaches rely on *structural metrics* [CK91], such as [BEG+06,LM06,SLT06] but some recent approaches are taking into account *semantic metrics* as well [DLOV08,ED00]. Structural metrics correspond to metrics derived from syntactic aspects of object-oriented code, such as the analysis of relationships among the methods and attributes of a class. The metrics defined by Chidamber and Kemerer [CK91] such as Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), and Coupling Between Objects (CBO) are typical examples of structural metrics. Semantic metrics are based on the analysis of the semantic information embedded

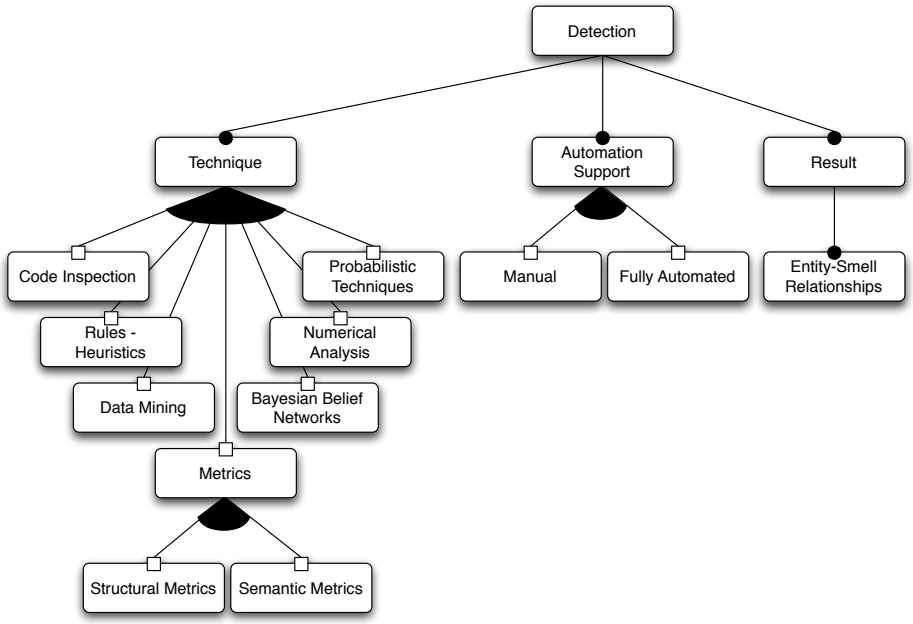

Fig. 8. Summary of the current situation for the detection activity.

in the code, such as comments and identifiers [ED00]. Knowledge-based, program understanding, and natural language processing techniques are used to compute such metrics. For example, the semantic LORM (Logical Relatedness of Methods) metric [ED00] measures the cohesion of a class, and more precisely the conceptual relatedness of the methods of the class, as determined by the understanding of the class methods represented by a semantic network of conceptual graphs. Several approaches use *rules or heuristic* knowledge to detect design smells [Ciu99,KRW07,LM06,MGDLM10]. Some approaches resort to more advanced techniques coming from the field of artificial intelligence, such as the use of *data mining* techniques [XS04]; from the probabilistic field, such as *Bayesian belief networks* [KVGS09]; or from *numerical analysis*, such as B-Splines [OKAG10]. However, these techniques may be insufficient to detect some code smells such as Shotgun Surgery [FBB+99] and Divergent Change [FBB+99], where the design change propagation probabilities between artefacts have to be considered when an artefact changes. Rao *et al.* [RR08] proposed a quantitative method for detecting these code smells using a *design change propagation probability* matrix.

Automation. Some of the proposed approaches are theoretical, aiming to get a scientific understanding of the intrinsic difficulties involved in detecting design smells. Other approaches are fully manual, such as the use of manual code inspection techniques to find design smells [TSFB99]. Most of the sur-

veyed approaches, however, provide explicit tool support, as is the case for [Chi02,Gro,MLC05,MGDLM10,Sli05,Tri08,TCC08,WP05].

Result. The detection activity produces, in all the the approaches we surveyed (for example in InCode-InFusion [Int08]), lists of *entity-smell relationships* for each detected smell. Those results are presented in a variety of textual and graphical ways.

Figure 8 summarises the techniques, automation support and results for the detection activity.

6.3 Visualisation

The *visualisation* activity produces some kind of graphical representation of the target artefact, allowing quick and easy identification of some of its properties. Many approaches address the visualisation activity in the context of design smell management, mostly to present detected smells in a graphical way. A visualisation tool can also provide other kinds of information. It can help the developer to decide which are the best modifications in order to remove a given smell. It can be used for evolution or for explaining the causes and impacts of smells. Visually summarising the properties and characteristics of the system and its parts can ease the realisation of any other activity.

Technique. The type of visualisation technique used mainly depends on the type of information that needs to be visualised. If this information is essentially a spreadsheet table, one can visualise it as pie *charts*, bar charts, line charts and the like. If the information is essentially a *graph*, one needs graph-based visualisation techniques and more or less sophisticated graph layout algorithms. This is the case, for example if one needs to represent (part of the) software structure, such as the dependency relationships [Con99,Tes08]. Some approaches [LM06] define new visualisation techniques, such as the *Overview Pyramid*. It is a metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and use of inheritance.

Automation. The activity of visualisation typically requires human intervention, either during the construction of the visual representation (e.g. by selecting the areas of interest, or choosing the most appropriate visual representation or layout algorithm) or during its use. If the visualisation is static, it can only be viewed [Con99,Tes08]. If it is dynamic, there is typically a direct and explicit link back to the target artefact under study (eg, InCode-InFusion [Int08]). This facilitates and speeds up detection and correction of design smells.

Result. Many of the surveyed approaches support some kind of visualisation by producing different types of diagrams to offer different types of general artefact visualisation in order to assist the comprehension of the target artefact. For example, in DECOR, systems are represented as class diagrams and classes infected by smells are highlighted in red. Other kinds of visualisation aids in design smell management include *entity-property visualisation* and *design smell*

visualisation [vM02] such as Overview Pyramid and Polymetric views [LM06] or dependency graphs [Con99].

Figure 9 summarises the techniques, automation support and results for the visualisation activity.

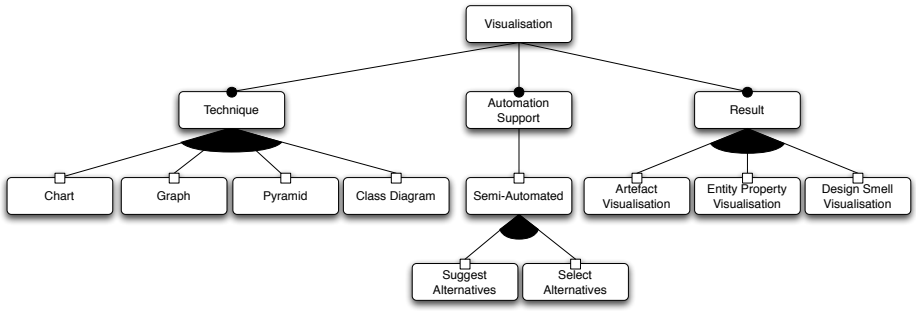

Fig. 9. Summary of the current situation for the visualisation activity.

6.4 Correction

Approaches supporting the *correction* of design smells, provide a way to (suggest how to) modify the target artefact, in order to remove smells and improve its design. During our survey, we found considerably less approaches supporting a (partially) automated correction of design smells, so this activity is clearly less mature than the smell detection or visualisation activity.

Technique. During our survey we have encountered mainly four different techniques for correcting design smells. The first technique is aimed at applying *rules* or *refactoring strategies* to design smells that have been previously detected [MGDLM10,TM03,TSG04]. These approaches have the advantage to provide a comprehensive process both for the detection and correction of smells. The second category suggests corrections (called *refactoring opportunities*) by relying only on metric values or the presence of certain patterns, without explicitly identifying design smells [BCT07,DLOV08,GAA01,SGM00,SSL01,SS07,TK04]. The third technique relies on ideas coming from the *machine learning* field, where *genetic algorithms* and other types of automated learning are exploited [BCT07,BAMN06]. Formal Concept Analysis (FCA) [GW99] is a fourth technique that has been intensively investigated for restructuring class hierarchies [ADN05,SLMM99,ST00] and classes affected by smells [MHVG08].

Automation. Various tools support smell correction in a semi-automated way, such as [BEG+06,TSG04,TCC08]. All of them need some additional interaction by the user.

Result. The correction of design smells can produce different types of results depending mostly on the degree of automation provided by each particular tool. Some tools can provide just *correction suggestions*, while others can produce some kind of *correction plans*, or specifications of the transformation sequence needed to improve the target artefact's design. For example, Trifu *et al.* [TSG04] proposed correction strategies mapping design smells to possible solutions. However, a solution is only an example of how the program *should have been implemented* to avoid a smell rather than a list of steps that a software engineer could follow to correct the smell.

Some tools can apply the computed changes to their internal artefact representation, therefore, producing a *transformed artefact model*. The fully-automated tools can operate straight over the target artefact, generating a *transformed artefact*.

Figure 10 summarises the techniques, automation support and results for the correction activity.

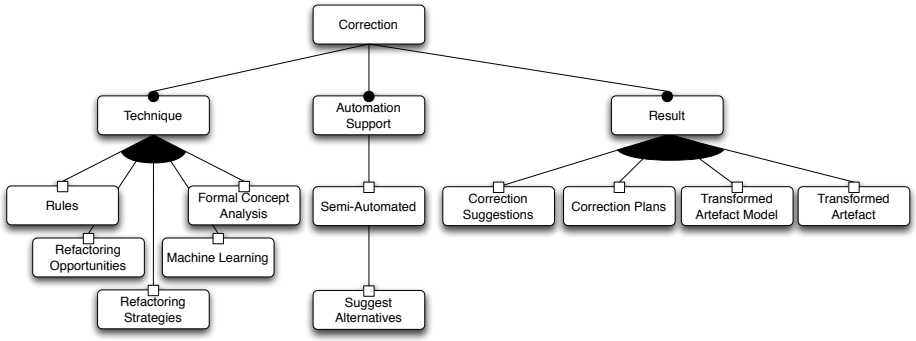

Fig. 10. Summary of the current situation for the correction activity.

6.5 Impact Analysis

The activity of *impact analysis* refers to the ability of an approach to compute the change impact of a design smell [vM02,VRDBDR07] or the actions performed to remove it [FTC07,LWN07b,TSG04].

Technique. This feature is offered mainly by approaches based on quality models [BAMN06,LM06,RSS+04,SGM00,TK04]. In [TK04], Tahvildari *et al.* present an approach based on soft-goal models. With soft-goal models they define the effects of design smells over metrics and system quality factors in a way that this information can be used automatically by a detection or a correction tool. Using soft-goal models the impact of design smells can be automatically computed to

assist the detection and correction activities. Marinescu introduces in [Mar02] how quality models can be used to estimate the impact of a design smell.

Deligiannis *et al.* [DSA+04] presented a controlled experiment on the impact of design smells, in which they studied 20 subjects to evaluate the impact of God Classes on the maintainability and understandability of object-oriented systems. The results of their study show that the Blob antipattern affects the evolution of design structures and the subjects' use of inheritance. Other similar approaches based on controlled experiments studied the impact of design smells on software quality factors such as comprehensibility [DDV+06] and maintenance [OCBZ09]. Some approaches used statistical models to investigate the relationship of design smells with class error probability [LS07] or with change-proneness [KPG09].

Recent works have studied the impact of design smells on software evolution by analyzing several versions of software systems [OCBZ09,VKMG09]. These approaches identify mainly evolution patterns of smells, which are then used to explain the impact of smells on the rest of the system. For example, Olbrich *et al.* [OCBZ09] analysed the historical data over several years of development of two large scale open source systems. They concluded that God Classes and Shotgun Surgery have a higher change frequency than other classes; and thus, may need more maintenance than non-infected classes.

Automation. Most of approaches evaluate the impact of design smells manually by conducting controlled experiments with subjects [DSRS03,DSA+04,DDV+06]. Some approaches integrate this activity in a fully automated way [TK04] to assist the detection and correction process.

Result. During our survey, we found that the most common results of impact analysis are the list of entities affected by a smell (*Entity Impact*) and the effect that a smell, or its removal, has over a metric value (*metric impact*) [DSRS03] or over a quality factor (*quality impact*) [DSA+04,DDV+06].

Figure 11 summarises the techniques, automation support and results for the impact analysis activity.

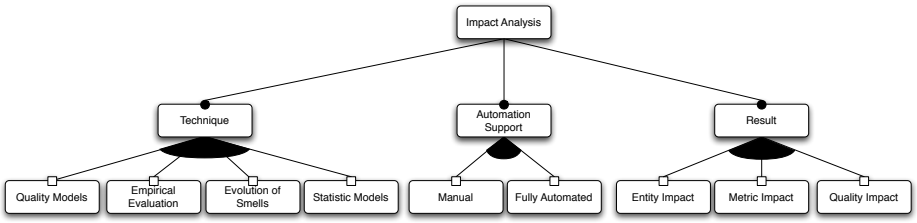

Fig. 11. Summary of the current situation for the impact analysis activity.

7 CONCLUSIONS AND FUTURE WORK

7.1 Open problems and future trends

Based on the findings of our survey, we discuss the remaining open problems in the field of design smell management, and present some avenues of future research to tackle these open challenges.

During our survey we have observed that not many of the studied approaches allow to reason about design smells at the level of design models (as opposed to source code and executable code). With the ever increasing importance of model-driven software engineering, it is imperative to have better future support for design smell management at the modeling level.

Related to the above challenge is the fact that almost none of the surveyed approaches is able to deal with design smells that involve different types of artefacts (e.g. a smell that involves both code, models and tests). Support for such types of smells will continue to gain importance, in the presence of multi-view and multi-language development environments.

Only a small fraction of the surveyed approaches took into account the version history. Such a rich data source is able to provide much more relevant information about why a particular design smell occurs, and how it may be corrected. As such, this can give rise to better and more reliable design smell detection and correction tools.

Most of the approaches we surveyed that support the activity of design smell correction were research prototypes. The next generation of commercial design smell management tools should therefore strive to integrate and automate correction techniques, rather than only supporting the detection activity.

Another challenge is to come up with better and more *language-agnostic* design smell management approaches. Most of the surveyed approaches focus on a specific programming language. A few of the tools, though, are applicable on more than one language.

As a final challenge, while the main purpose of current-day design smell management approaches is to improve the *software product quality* (by detecting and correcting "smelly" parts of the software), they could also be used to improve the *software process quality*. In many situations, the cause of a design smell may be a suboptimal software process. (For example, if the software process does not discourage copy-paste reuse, the software is likely to suffer from code duplication and high coupling between modules. Similarly, if the process does not encourage modular design, the software is likely to suffer from cyclic dependencies.) Hence, the smell correction activity should not only suggest to correct the detected problem itself, but also its cause, by providing concrete suggestions on how to improve the software process to avoid introducing design smells before they occur.

7.2 Conclusion

In this technical report we presented a structured taxonomy of object-oriented design smell management approaches, using the technique of feature modelling.

Our taxonomy allows researchers, developers and tool vendors to: (1) position concrete approaches within the domain; (2) use the proposed framework for comparing and combining individual tools and approaches; (3) identify and evaluate tools for a specific design smell management activity; (4) provide an overview of the research field of design smell management; (5) identify open problems and challenges and suggest new avenues of research.

As part of our survey, we have proposed a unifying terminology for design smells. This can be helpful for researchers in this field, since design smells have been given many different names and have been described in many different ways. We hope that this unifying terminology will by adopted by tool developers in order to avoid any future confusion or misunderstanding, and to facilitate communication and tool comparison.

8 ACKNOWLEDGEMENTS

Javier Pérez and Carlos López are partially funded by the Spanish government (*Ministerio de Ciencia e Innovación*, project TIN2008-05675). Tom Mens is supported by the F.R.S.-FNRS through FRFC project 2.4515.09; by ARC project AUWB-08/12-UMH financed by the *Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique*; and by FEDER portefeuille TIC - project CEIQS financed by the Walloon Region.

References

[ADN05] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In 9^{th} *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 62–71, 2005.

[Ana08] Analyst4j. http://www.codeswat.com, February 2008. [Accessed: 2010-04-06].

[Aqr02] Aqris. RefactorIT. http://www.refactorit.com, 2002. [Accessed: 2010-04-06].

[Arg] ArgoUML. http://argouml.tigris.org. [Accessed: 2009-10-23].

[AS09] El Hachemi Alikacem and Houari A. Sahraoui. A metric extraction framework based on a high-level description language. In 9^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 159–167, Washington, DC, USA, 2009. IEEE Computer Society.

[BAMN06] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. In 8^{th} *annual conference on Genetic and evolutionary computation*, pages 1885–1892, New York, NY, USA, 2006. ACM.

[BCT07] Thierry Bodhuin, Gerardo Canfora, and Luigi Troiano. Sormasa: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In 1^{st} *Workshop on Refactoring Tools* [DC07], pages 23–24.

[BEG+06] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiss. Trex - the refactoring and metrics tool for ttcn-3 test specifications. In *Testing: Academic and Industrial Conference - Practice And Research Techniques*, pages 90–94, Aug. 2006.

[BGQR07] Giulia Bruno, Paolo Garza, Elisa Quintarelli, and Rosalba Rossato. Anomaly Detection in XML Databases by means of Association Rules. In 18^{th} *International Conference on Database and Expert Systems Applications*, pages 387–391, Washington, DC, USA, 2007. IEEE Computer Society.

[BMMIM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, March 1998.

[Bor] Borland. Together. http://www.borland.com/us/products/together. [Accessed: 2010-04-06].

[Bro75] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Reading, MA , USA, 1975.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Feature Modeling*, chapter 5, pages 83–116. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, June 2000.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Che04] CheckStyle. http://checkstyle.sourceforge.net, 2004. [Accessed: 2010-04-06].

[Chi02] Ciprian-Bogdan Chirila. Automation of the design flaw detection process in object-oriented systems. *International Conference on Technical Informatics (CONTI); Periodica Politechnica – Transactions on Automatic Control and Computer Science*, 47, October 2002.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *International Conference on Technology of Object-Oriented Languages and Systems*, pages 18–32, Washington, DC, USA, 1999. IEEE Computer Society.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices*, 26(11):197–211, 1991.

[CLMM06] Yania Crespo, Carlos López, Esperanza Manso, and Raúl Marticorena. From bad smells to refactoring, metrics smoothing the way. In *Object-Oriented Design Knowledge. Principles, Heuristics and Best Practices*, Object-Oriented Design Knowledge. Principles, Heuristics and Best Practices, chapter VII, pages 193–249. Idea Group Publishing, 2006.

[Con99] Clarkware Consulting. JDepend. http://clarkware.com/software/JDepend.html, 1999. [Accessed: 2010-04-06].

[Cre00] Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones*. PhD thesis, Universidad de Valladolid, 2000.

[DC07] Danny Dig and Michael Cebulla. 1^{st} workshop on refactoring tools. Technical Report 2007-8, TU Berlin, July 2007.

[DDV+06] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355. IASTED/ACTA Press, 2006.

[DLOV08] Andrea De Lucia, Rocco Oliveto, and Luigi Vorraro. Using structural and semantic metrics to improve class cohesion. In *International Conference on Software Maintenance*, pages 27–36. IEEE, October 2008.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, pages 109–118, 1999.

[DSA+04] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, July 2004.

[DSRS03] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003.

[Dur07] Nakul Durve. Coderaider: Automatically improving the design of code. diploma thesis, June 2007.

[Ecl] Eclipse Metrics Plugin. http://eclipse-metrics.sourceforge.net. [Accessed: 2010-04-06].

[ED00] L. Etzkorn and H. Delugach. Towards a semantic metrics suite for object-oriented design. In 34^{th} *International Conference on Technology of Object-Oriented Languages and Systems*, pages 71–80, 2000.

[EGR+06] Janees Elamkulam, Ziv Glazberg, Ishai Rabinovitz, Gururaja Kowlali, Satish Chandra Gupta, Sandeep Kohli, Sai Dattathrani, and Claudio Paniagua Macia. Detecting design flaws in uml state charts for embedded software. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2006.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, June 1999.

[FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of feature envy bad smells. In *International Conference on Software Maintenance*, pages 519–520. IEEE, Oct. 2007.

[FXC06] FXCop. http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx, June 2006. [Accessed: 2010-04-06].

[GA08] Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A Multilayered Framework for Design Pattern Identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, September/October 2008.

[GAA01] Yann-Gaël Gueheneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. 39^{th} *International Conference on Technology of Object-Oriented Languages and Systems*, pages 296–305, 2001.

[GDMR04] Tudor Girba, Stéphane Ducasse, Radu Marinescu, and Daniel Ratiu. Identifying entities that change together. In 9^{th} *IEEE Workshop on Empirical Studies of Software Maintenance*, Chicago, 2004.

[GFGP06] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In 9^{th} *International Conference of Funtamental Approaches to Software Engineering*, number 3922 in Lecture Notes in Computer Science, pages 411–425. Springer, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[Gro] LOOSE Research Group. iPlasma. http://loose.upt.ro/iplasma. [Accessed: 2010-04-06].

[Gué03] Yann-Gaël Guéhéneuc. *A framework for design motif traceability*. PhD thesis, École des Mines de Nantes; University of Nantes, July 2003.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin/Heidelberg, 1999.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[Ham07] Hammurapi. http://www.hammurapi.biz, October 2007. [Accessed: 2010-04-06].

[Int08] Intooitus. Incode infusion. http://www.intooitus.com/, 2008. [Accessed: 2010-04-06].

[ISO01] ISO and IEC. ISO/IEC 9126-1:2001, software engineering – product quality, part 1: Quality model, 2001.

[JFRS07] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. In 33^{rd} *International Conference on Very Large Databases*, pages 1263–1274. VLDB Endowment, 2007.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Signature Series. Addison-Wesley Professional, August 2004.

[KPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In 16^th *Working Conference on Reverse Engineering (WCRE)*, pages 75–84. IEEE Computer Society, 2009.

[KRW07] Douglas Kirk, Marc Roper, and Murray Wood. A heuristic-based approach to code-smell detection. In 1^{st} *Workshop on Refactoring Tools* [DC07], pages 55–56.

[KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari A. Sahraoui. A bayesian approach for the detection of code and design smells. In *International Conference on Quality Software*, pages 305–314. IEEE Computer Society, 2009.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[LS07] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, July 2007.

[LWN07a] Ángela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Assessing the impact of bad smells using historical information. In 9^{th} *International Workshop on Principles of Software Evolution*, pages 31–34, New York, NY, USA, 2007. ACM.

[LWN07b] Ángela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In 4^{rd} *International Workshop on Mining Software Repositories*, May 2007.

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, October 2002.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.

[MGDLM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January/February 2010.

[MGLM+09] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, May 2009.

[MHVG08] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In Raoul Medina and Sergei Obiedkov, editors, 4^{th} *International Conference on Formal Concept Analysis*, pages 289–304. Springer-Verlag, February 2008.

[MLC05] Raúl Marticorena, Carlos López, and Yania Crespo. Parallel inheritance hierarchy: Detection from a static view of the system. In *6th International Workshop on Object Oriented Reenginering (WOOR), Glasgow, UK.*, page 6, July 2005. http://smallwiki.unibe.ch/woor/workshopparticipants/.

[MLC06] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In 7^{th} *International Workshop on Object-Oriented Reengineering*, page 6, Nantes, France, July 2006.

[Mun05] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. *Software Metrics, 2005. 11th IEEE International Symposium*, pages 9 pp.–, Sept. 2005.

[MVDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July/August 2005.

[NB07] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In 19^{th} *IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software*, volume 4581/2007 of *Lecture Notes in Computer Science*, pages 228–243. Springer, Heidelberg, June 2007.

[OCBZ09] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In 3^{rd} *International Symposium on Empirical Software Engineering and Measurement*, ESEM'09, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.

[OKAG10] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In 14^{th} *European Conference on Software Maintenance and Reengineering*, pages ??–?? IEEE Computer Society, March 2010.

[PMD09] PMD. http://pmd.sourceforge.net, June 2009. [Accessed: 2010-04-06].

[PP07] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Raţ04] Daniel Raţiu. Memoria: A Unified Meta-Model for Java and C++. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, 2004.

[Rev] RevJava. http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm. [Accessed: 2010-04-06].

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 1996.

[Roo] Roodi. http://roodi.rubyforge.org. [Accessed: 2010-04-06].

[RR08] A. Ananda Rao and K. Narendar Reddy. Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, March 2008.

[RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *International workshop on Mining Software Repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

[RSS+04] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: Smart analysis based error reduction. *ACM SIGSOFT Software Engineering Notes*, 29(4):243–251, 2004.

[Rut] Kevin Rutherford. Reek. http://wiki.github.com/kevinrutherford/reek. [Accessed: 2010-04-06].

[Sem07] SemmleCode. http://semmle.com/semmlecode, October 2007. [Accessed: 2010-04-06].

[SGM00] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *International Conference on Software Maintenance*, pages 154–162, Washington, DC, USA, 2000. IEEE Computer Society.

[She00] Thomas B. Sheridan. Function allocation: algorithm, alchemy or apostasy? *Int. J. Hum.-Comput. Stud.*, 52(2):203–216, 2000.

[Sli05] Stefan Slinger. Code smell detection in eclipse. Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science - Delft University of Technology, 2005.

[SLMM99] Houari A. Sahraoui, Hakim Lounis, Walcélio L. Melo, and Hafedh Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering*, 6(4):387–410, 1999.

[SLT06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A metric-based heuristic framework to detect object-oriented design flaws. In 14^{th} *IEEE International Conference on Program Comprehension*, 2006.

[SR06] Martin Lippert Stefan Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2006.

[SS07] G. Snelting and M. Streckenbach. Kaba: Automated refactoring for improved cohesion. In 1^{st} *Workshop on Refactoring Tools* [DC07], pages 1–2.

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In 5^{th} *European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.

[ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions Programming Languages Systems*, 22(3):540–582, 2000.

[STA] STAN. http://stan4j.com. [Accessed: 2010-04-06].

[Sty] StyleCop. http://code.msdn.microsoft.com/sourceanalysis. [Accessed: 2010-04-06].

[TCC08] Nikolaos Tsantalis, Tsantalis Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In 12^{th} *European Conference on Software Maintenance and Reengineering*, pages 329–331, April 2008.

[Tea09] Ptidej Team. DECOR. http://www.ptidej.net/research/decor, 2009. [Accessed: 2010-04-06].

[Tes08] Jean Tessier. Dependency Finder. http://depfind.sourceforge.net, 2008. [Accessed: 2010-04-06].

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

[TK04] Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 16(4-5):331–361, 2004.

[TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In 7^{th} *European Conference on Software Maintenance and Reengineering*, pages 91–100, Washington, DC, USA, 2003. IEEE Computer Society.

[TMMM07] Alban Tiberghien, Naouel Moha, Tom Mens, and Kim Mens. Répertoire des défauts de conception. Technical Report 1303, University of Montreal, 2007.

[Tri08] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.

[TSFB99] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In 14^{th} *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and applications*, pages 47–56, New York, NY, USA, 1999. ACM.

[TSG04] Adrian Trifu, Olaf Seng, and Thomas Genssler. Automated design flaw correction in object-oriented systems. In 8^{th} *European Conference on Software Maintenance and Reengineering*, pages 174–183. IEEE Computer Society Press, March 2004.

[VKMG09] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE'09)*, pages 145–154, Los Alamitos, CA, USA, October 2009. IEEE Computer Society.

[vM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In 9^{th} *Working Conference on Reverse Engineering*, pages 97–107. IEEE Computer Society, October 2002.

[vMvK01] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In 2^{nd} *International Conference on Extreme Programming*, 2001.

[VRDBDR07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

[Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Web95] Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, February 1995.

[WP05] Bartosz Walter and Blazej Pietrzak. Multi-criteria detection of bad smells in code with UTA method. In 6^{th} *International Conference on Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *Lecture Notes in Computer Science*, pages 154–161. Springer, June 2005.

[XS04] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In 16^{th} *International Conference on Software Engineering & Knowledge Engineering*, pages 123–128, 2004.