Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Towards a Principle-based Classification of Structural Design Smells

S G Ganesh[a] Tushar Sharma[a] Girish Suryanarayana[a]

a. Siemens Corporate Research & Technologies, #84, Keonics Electronic City, Hosur Road, Bangalore - 560 100, India.

Abstract Fred Brooks in his book "The Mythical Man Month" describes how the inherent properties of software (i.e. complexity, conformity, changeability, and invisibility) make its design an "essential" difficulty. Good design practices are fundamental requisites to address this difficulty. One such good practice is that a software designer should be aware of and address "design smells" that can manifest as a result of his design decisions. However, our study of the vast literature on object-oriented design smells reveals the lack of an effective organization of smells that could better guide a designer in understanding and addressing potential issues in his design. In order to address this gap, we have adopted a novel approach to classify and catalog a number of recurring structural design smells based on how they violate key object oriented (OO) design principles. To evaluate the usefulness of our design smell catalog, we first asked Siemens CT DC AA architects to use it to identify design smells in their projects, and later elicited feedback from them about their experience. The feedback received indicates that these architects found the catalog to be very useful. In this paper, we present our catalog, classification, and naming scheme for design smells and also highlight several interesting observations and insights that result from our work.

Keywords Design smells, design principles, object oriented design, design smell classification, design smell template

1 Introduction

Design smells are often introduced unintentionally by practitioners during software development. For instance, a software designer may adopt well-known established practices during initial design; however, the emergent design may indicate certain structural deficiencies or smells that have inadvertently cropped up during the process. Similarly, software developers who are tasked with maintaining a piece of software (i.e.

S G Ganesh, Tushar Sharma, Girish Suryanarayana. *Towards a Principle-based Classification of Structural Design Smells*. In Journal of Object Technology, vol. 12, no. 2, 2013, pages 1:1–29.
doi:10.5381/jot.2013.12.2.a1

develop new features or fix bugs in a short time period) may unknowingly introduce smells in the design. It is important in both cases to be aware of and address the smells so as to reduce the technical debt [Cun92] and maintain a high structural quality of the software.

The importance of smells and the role they play in the practice of software development is clearly reflected in the large amount of literature that exists on design smells. However, a study of this literature reveals a few shortcomings. First, a single yet reasonably comprehensive catalog of recurring design smells is missing. Such a catalog can serve as a quick reference for designers so that they can recognize, avoid, and even address potential problems in their design. Second, very little work has been done on classifying these smells. Further, the few classification schemes that exist for categorizing smells lack consistency and are not very useful with respect to providing guidance to designers. Providing a consistent classification scheme aids practitioners to get a coherent view of design smells. Further, we believe that a classification scheme will be more useful when it provides high-level hints on understanding the cause of the smell as well as what needs to be done to fix the smell. Third, a consistent and simple naming scheme for design smells is missing in the literature. Following a consistent and simple naming scheme for design smells is useful since it can provide a common vocabulary for discussing design smells.

This paper describes our efforts towards addressing these shortcomings. Towards the first shortcoming, we have embarked on creating a collection of documented design smells. However, given the large number of documented smells and their varying ambit, we realized early on the need to scope our efforts in order to provide a focused catalog meaningful for practitioners. We have, therefore, in this paper, limited our catalog to structure-related smells that are found at the micro-architectural level. Currently, our catalog only includes smells that have been documented in literature.

Towards the second shortcoming, we have attempted to categorize design smells using a novel classification scheme. Our analysis of smells has shown that the cause of all the smells in our catalog can be traced to the violation of fundamental design principles. In other words, there is an association between a principle and a smell. Our classification scheme, therefore, categorizes design smells in our catalog using violations of these design principles as the basis. We believe that since our classification scheme explicitly links the cause of a smell to a principle, it will better guide a designer in addressing that smell.

Towards addressing the third shortcoming, we have leveraged our principle-based classification scheme to create a new naming scheme for smells. This scheme assigns a name to a smell that captures both the main design principle that was violated along with a characterization of how that smell violates the main design principle. This naming scheme has two main benefits. First, since the name of a smell explicitly indicates the fundamental design principle that was violated, it provides a hint to the designer about what was possibly done wrong during the design process and therefore what must be done to address the issue. Second, it serves as an excellent basis to aggregate similar smells with varying names under one unifying umbrella.

To evaluate the usefulness of our design smells catalog, classification, and naming scheme, as a first step, we asked 12 Siemens CT DC AA architects to use our catalog to identify design smells in their projects. The feedback from this initial evaluation from practitioners clearly shows that the design smells catalog has helped them to broaden their knowledge of design smells, better understand causes of design smells, and serve as a quick reference guide in their daily work.

In summary, the main contributions of our paper (supported by initial evaluation in industry context) are:

- A catalog consisting of 31 design smells with references to 210 related design smells in the literature. To the best of our knowledge, this is the most comprehensive catalog available in literature on structural design smells so far.
- A classification scheme where design smells are classified as violation of one of the four fundamental OO design principles (principles of abstraction, encapsulation, modularity, and hierarchy).
- A concise and simple naming scheme for design smells.

The rest of the paper is organized as follows. Section 2 describes the motivation and background of our work. Section 3 introduces our novel classification scheme for design smells. Section 4 describes the template that we have used to document smells in our catalog. Section 5 presents our catalog based on our principle-based classification. Section 6 briefs the results of the initial evaluation of our design smells work. Section 7 summarizes the observations and insights that have resulted from our categorization work and Section 8 lists some of the ways we plan to extend our work in the future.

2 Motivation and Background

Before we delve into the shortcomings of available literature on smells, it is useful to discuss the various connotations of the word "smell". The word smell in the context of software design has been defined differently by different people. Some well-known definitions of smells are as follows:

- "We suggest that, like any living creature, system designs are subject to diseases, which are design smells (code smells and anti-patterns). Design smells are conjectured in the literature to impact the quality and life of systems." [HKGH10]
- "Smells are certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring." [Fow99]
- "Code and design smells are poor solutions to recurring implementation and design problems." [MGDM10]
- "Design smells are the odors of rotting software." [Mar03]

Based on the above definitions, it can be seen that while some treat a smell as a problem itself, others consider a smell to be only indicative of a deeper problem. One of the primary objectives of our classification and catalog is to create a framework to categorize all documented smells and to organize similar smells under a single canopy. Hence, our catalog embraces all these above definitions and often groups together smells that may fit different definitions but still relate to a similar violation of a fundamental design principle.

Even before we begin a discussion on the limitations of existing works on design smells, it is insightful to first reflect on the varying terminologies that have been used to indicate design smells. A non-exhaustive list of such terms (by looking from different levels of abstraction and incorporating different perspectives) cited

in literature includes: "bad smells" [Fow99], "violation of design heuristics" [Rie96], "design principle violations" [Mar02], "design smells" [MGDM10], "indicators of design problems" [Eic03], "(design) problem symptoms" [PJ99], "design pattern defects" [MGDM10], "design flaws" [TM05], "design disharmonies" [LM06], "danger signals" [Mey00], "change smells" [RFG05], "problem patterns" [SSM06], "structural anomalies" [TM05], "structural anomalies and problems" [TM05], "architectural smells" [LR06], and "anti-patterns" [BMB+98].

Given the fact that so many terms have been used to denote design smells, it is not surprising to find that a similar large body of (possibly overlapping) knowledge exists for smells in the literature. However, these suffer from the three main shortcomings, which we discuss next.

2.1 Lack of Well-Organized Catalogs

Many attempts have been made to collect and present design smells; however, in most cases there is no clear separation of design smells at different levels of abstractions. There exist catalogs that document design smells that pertain to the overall architecture of the system together with problems that relate to the implementation level. An example is the catalog of 52 "problem patterns" offered by Frank et al. [SSM06]. It covers a wide range of problem patterns:

- architectural level problem patterns such as "god package" and "cyclical dependency between packages"
- micro-architectural patterns such as "knows of derived" and "polymorphism placebo"
- implementation level problems such as "improper name length" and "variables having constant value"

An architectural issue clearly has different concerns and targets a different audience from that of an implementation issue. Therefore, the lack of scope-based organization limits the usefulness of such catalogs. Further, several known catalogs of design smells, for example, catalog of design rules by Garzas et al. [Gar07], bad smells by Fowler [Fow99], problem patterns by Frank et al. [SSM06] fail to provide a coherent structure around which these problems can be organized. This tends to limit the usefulness of these catalogs as a reference guide for designers and developers.

2.2 Lack of Consistent and Useful Classification Schemes

Providing a consistent classification scheme aids practitioners to get a coherent view of design smells. Further, we believe that a classification scheme will be more useful when it provides high-level hints on understanding the cause of the smell as well as what needs to be done to fix the smell. In this section, we briefly outline existing smell catalogs that present an organization of smells around a classification scheme and discuss their shortcomings.

- The inFusion tool [InF12] classifies "design flaws" based on the following "design-properties": size and complexity, encapsulation, coupling, cohesion, and hierarchy. Choinzon and Ueda [CU06] classified design defects into ten categories. Both of these categorizations suffer from inconsistent classification schemes: The

categories are a mix of principles (such as "encapsulation"), language features (such as "inheritance"), and measures (such as "size" and "complexity").

- Mantyla [MVL03] classified Fowler's code smells [Fow99] as "encapsulators", "bloaters", "couplers", "OO abusers", "change preventers", "dispensables", and "others". This classification scheme has numerous shortcomings: it lacks maturity (since even in the small set of Fowler's 22 smells, two of them are classified in the "others" category) and inconsistent (for instance, the term "encapsulators" does not have a negative connotation unlike other categories such as "bloaters" and "couplers").

- Wake [Wak03] extended Fowler's list of code smells [Fow99], and categorized them at a high level as smells within classes and smells between classes. He further classified smells into categories such as "library classes", "data", "names", and "accommodating change". The second level categories are inconsistent and do not provide a collective coherent meaning.

- Moha et al. [MGDM10] classified problems into intra-class and inter-class problems at the top level. At the next level, Moha's classification categorized problems as structural, lexical, and measurable. The second level categorization is based on the techniques used to detect the smells, namely static analysis, natural language processing, and metrics. While this categorization is consistent from the perspective of how to detect these smells, they are not very useful to understand the cause of the smell or for ways to address them.

2.3 Lack of Common Consistent Vocabulary

During our extensive literature survey, we did not come across any attempt at creating a uniform naming scheme for design smells. Creating a consistent and simple naming scheme for design smells is useful since it can provide a common and easy to use vocabulary for discussing design smells. We believe that such a naming scheme is especially important given the fact that hundreds of design smells are documented in literature (see Section 5).

3 Principle-based Classification Scheme for Smells

As described in the previous section, most existing catalogs of design smells lack a consistent and useful organization of problems. In order to address this, we have attempted a novel classification scheme for smells. In this section, we first describe the scope of our work followed by our proposed classification scheme.

3.1 Limiting Scope to Structural Micro-architectural Smells

Bad smells can be classified into architectural, design (i.e. micro-architectural), and implementation level smells based on the granularity of abstraction. These smells can also be characterized as creational, structural, and behavioral in nature. In this paper, we limit our discussion to design-level structural smells (highlighted with • symbol in Table 1). Henceforth, we use the concise term "design smell" to mean "micro-architectural-level structural smells" in this paper. It should be noted that the design smells cataloged in this paper have not been invented by us; we believe that just like

		Characterization		
		Creational	Structural	Behavioral
Granularity	Architectural			
	Design		•	
	Implementation			

Table 1 – Scope of smells presented in this text

design patterns [GHJV95], design smells are also "an aggressive disregard of originality" [MRB97]. The cataloged smells are strictly those that have been documented earlier in the literature.

3.2 Classification Based on Violation of Design Principles

Our reflection on the problems with existing classification schemes for design smells revealed the need for a more practical and useful classification scheme. Specifically, we identified the following objectives for our classification scheme:

- It should enable a hierarchical, uniform, and technical categorization of smells.
- The names of the smells itself should indicate the affected design principles:
  - to enable an intuitive understanding of the smell.
  - to guide the adoption of suitable measures to address the smell.

3.2.1 First Level of Classification

Towards addressing these objectives, we realized that a classification approach grounded on fundamental design principles (relevant to object orientation) would be more helpful to practitioners. If a practitioner could easily trace the cause of smells to violated design principles, it would better guide him in addressing that smell.

There are many related works in the area of software design that explicitly refer to identifying violation of design principles. For example, Langelier et al. [LSP05] discusses how an expert can estimate whether a portion of the code violates the well-known principle of "low coupling and high cohesion" using their visualization framework. However, we haven't come across any comprehensive approach for classification of design smells based on violation of fundamental design principles as the basis.

To create such a classification model, we studied a number of well-known principles, guidelines, and approaches for software design documented in literature. From the available works, we found the "object model" (which is a conceptual framework of object orientation) proposed by Booch et al. [BME+07] to be most suitable for our needs. The four major elements of the object model are:

Abstraction: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

Encapsulation: "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

Modularity: "Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

Hierarchy: "Hierarchy is a ranking or ordering of abstractions."

We chose these four principles as the basis for our classification scheme for the following reasons:

- These major principles "abstraction", "encapsulation", "modularity", and "hierarchy" are well-known and fundamental design principles. Thus, a classification based on these principles would be familiar and easy to remember for practitioners.
- All design smells in our catalog could be traced to violations of the four principles.
- A concise classification (with only four highest-level entities) provides the advantage that it is easier to remember than a classification that has a lengthy list of high-level categories.
- Each principle is a single term and can be prefixed by a suitable qualifying adjective to name a design smell. The resulting term (i.e. adjective + principle) thus can denote violation of a principle, and guide refactoring to address that smell.

It should be noted that Booch et al.'s definition of modularity refers to the term "module" which is typically used in the context of high-level (or architectural) design. However, since our discussion in this paper is limited to micro-architectural smells, we have used the principle of "modularity" strictly in the context of class-level abstractions (and their relationship with other abstractions at the same level of granularity).

We observed that most design smells in our catalog can be traced to the violation of one of the four design principles and it was easy to classify these smells under that principle. We also observed that some smells can be caused due to the violation of more than one design principle. This is because these design principles are not mutually exclusive. For usability reasons, in our catalog, a smell is classified under and named by the principle which is violated most by the smell. For instance, the "envious abstraction" smell, as defined by our catalog, arises when members of an abstraction are more interested in other abstractions leading to increased coupling. Thus, both the principles of abstraction and modularity are violated. However, since this smell arises primarily due to wrong assignment of responsibilities across abstractions, it mainly affects abstraction and is therefore categorized under "abstraction".

3.2.2 Second Level of Classification

After we cataloged a set of documented smells under a particular principle, we proceeded with creating a second-level classification. For this, we carefully studied the set of smells categorized under that principle and identified how a particular smell or its variant primarily manifests. Next, we grouped similar primary manifestations under the canopy of a single smell. For instance, consider the specific set of smells under hierarchy that manifest in the form of classes having a number of concrete methods which are not overridden by subtypes. These smells indicate an aberration in the application of inheritance in order to achieve implementation reuse. Thus, this set of smells was classified under "convenience hierarchy" (see Section 5.4.5). Additionally, a constraint for the second-level classification was to maintain a fine-grained granularity

for the categories. This was essential in order to keep the classification simple and allow a separate discussion for each smell, including the most basic ones. Figure 2 provides an overview of our proposed classification scheme.

3.3 Concise and Uniform Naming Scheme

We have leveraged our classification scheme to attempt a new naming scheme for structural design smells. The name for each design smell consists of two words: an adjective (the first word) that qualifies the name of the violated design principle (the second word). This results in concise names for design smells which aids easy recall. It also allows the cause of a smell to be traced back to a violated design principle thus guiding designers in adopting a suitable solution to address those smells.

4 Documentation of Cataloged Smells

In order to properly document the smells in our catalog so that they can be useful to practitioners, we have created a detailed template that captures their important characteristics. Table 2 shows the elements of the template. In this section, we present one of the smells in our catalog that has been documented using this template to illustrate the usefulness of this detailed smell template. Due to space constraints, we present these smells in a condensed format in Section 5. We are currently in the process of publishing the detailed catalog as a book.

4.1 An Example: Cyclic Hierarchy

Name: Cyclic hierarchy.

Short description: This design smell arises when a supertype directly or indirectly refers to one of its subtypes, forming a cycle in the hierarchy [Ber04, BNL05, Gar07, Rie96, DMTS10a, WCKD11].

Long description: This smell arises when a supertype refers to any of its subtypes. Here the term "reference" means: (a) a supertype contains an object of one of its subtypes (b) a supertype refers to the type name of one of its subtypes (c) a supertype accesses data members, or calls methods from one of its subtypes. Such a reference can be either direct or indirect. Since subtype knowledge introduces a cycle in the inheritance graph annotated with class relationships, this smell is termed as "cyclic hierarchy".

Rationale: It is undesirable for a supertype to have knowledge about its subtype(s) for the following reasons:

- Supertypes and subtypes can be thought of as separating the interface and the implementation aspects of an abstraction. An interface should specify a contract and should not know anything about the implementation, and hence a supertype should not have any knowledge of its subtypes.
- A supertype needs to be agnostic of any of its subtypes; only then, it is possible to compile it, use it and reuse it independent of its subtypes. With subtype

Element	Short description
Name	A concise, intuitive name based on our naming scheme described in Section 3.3.
Short description	A short description of the design smell to explain the smell concisely.
Long description	A detailed description about the design smell to explain the design smell (with optional discussion on potential causes of the smell, or its effects on design).
Rationale	A rationale to justify why the identified issue indicates a bad smell in design.
Example(s)	One or more illustrative examples preferably from well-known open source software.
Violated principles	OO design principles (out of the four principles - abstraction, encapsulation, modularity, and hierarchy) whose violations can lead to this smell, the specific object oriented enabling techniques that have been violated, and the justification for why the design smell is classified under a particular principle.
Impacted quality attributes	The design quality attributes (reusability, flexibility, understandability, functionality, extendability, and effectiveness [BD02]) that are negatively impacted because of this smell.
Also known as	Alternative names documented in literature that are used to describe the design smell.
Variants	Design smells documented in literature that are fundamentally identical to and yet exhibit a slight variation from the smell. The variation may include a special form, or a more general form of the design smell.
Exceptions	Contexts or situations where the smell is likely to be not considered a problem.
Detection strategy	High-level hints on how the design smell can be detected from design or code artifacts.
Suggested refactoring	Generic high-level suggestions and steps to refactor the design smell.

Table 2 – Design smell template

knowledge, changes to subtypes can potentially affect the supertype, which is undesirable.

Example(s): The example in Figure 1 shows inheritance relationship between `MutableBigInteger` and `SignedMutableBigInteger` from JDK; these classes were introduced in Java from version 1.3 and are part of `java.math` package. The private `modInverse()` method in `MutableBigInteger` class implements an algorithm that requires the use of signed integer and hence it creates instances of `SignedMutableBigInteger`; this instance creation introduces a reference from the supertype method to the subtype resulting in cyclic hierarchy smell. One potential refactoring for addressing this smell is to re-implement the `modInverse()` method (say, using an alternative algorithm if available) in such a way that it does not require creating instances of `SignedMutableBigInteger`.

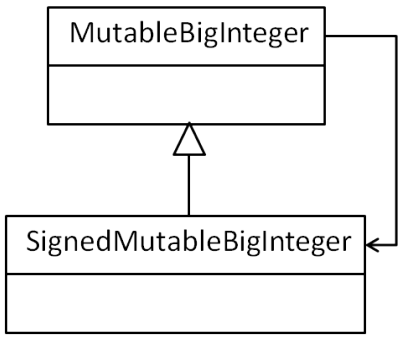

Figure 1 – An example of cyclic hierarchy smell from java.math library

Violated principles:

Abstraction: A forward reference to the subtype likely indicates ineffective abstraction of the problem domain [Mil99]. Thus the principle of abstraction is violated.

Modularity: Cyclic dependencies create tight coupling and thus violate the principle of modularity. Specifically, cyclic dependencies indicate the violation of the Acyclic Dependencies Principle (ADP) [Mar03]. Further, changes to subtype might require re-compilation of the supertype affecting independent module compilability [SRK07].

Hierarchy: Ideally, inheritance should model a generalization/specialization hierarchy and the hierarchy specializations should have dependencies that are directed towards generalizations. A forward association from the supertype to subtype violates Dependency Inversion Principle [Mar03], and this changing of dependency direction towards specializations violates the principle of hierarchy.

This smell can only manifest in the context of a hierarchy. We, therefore, classify it under the principle of hierarchy.

Impacted quality attributes:

Flexibility: Changes to or removal of subtypes affect supertype, hence flexibility is impacted.

Understandability: In order to understand the supertype, one has to understand the subtype also. This increases the cognitive load, affecting understandability.

Extendability: Adding a new subtype may require changes in supertype, thus impacting extendability of the design.

Also known as: "Knows of derived" [Ciu99, SSM06], "subtype knowledge" [DMTS10b], "subclass knowledge" [BNL05], "curious superclasses" [BNL05], "inheritance/reference cycles" [SSC96], "base class depends on derived classes" [CWZ00], "forward associations in a hierarchy" [Mil99].

Variants: "Inheritance loops" [Bin99].

Exceptions: This design smell might be acceptable if it is known that the supertype and subtype(s) are not going to change in future [SSC96]. Sometimes, to support

unanticipated changes, when inheritance is used as registration mechanism or factory where the client should always refer to the supertype, it is acceptable for supertypes to refer to their subtypes. Examples (both from [DMTS10a]): Returning instance of a subtype as the default instance in Singleton pattern and installing global default factory in case of Abstract Factory pattern.

Detection Strategy: Check if the inheritance graph annotated with class relationships is a Directed Acyclic Graph (DAG).

Suggested refactoring: If the reference from supertype to subtype is accidental or incidental, refactor the supertype to eliminate such references. If the supertype requires the services of its subtypes, consider applying State or Strategy patterns [Gar07].

5 Design Smell Catalog Using Principle-based Classification

This section presents our design smell catalog which has been created by categorizing an extensive set of documented structural design smells using our principle-based classification scheme. Our catalog is well-organized and avoids the problem of varying scope by focusing on structural micro-architectural smells only (see Table 1). It consists of 31 design smells with references to 210 related design smells, and is by far the most comprehensive design smells catalog in literature.

However, it should be noted that there are certain smells that are not a part of our catalog. For instance, we have not included smells that have extremely long names such as Riel's heuristics [Rie96] which describe many potential design smells verbosely in a non-concise fashion. Further, we have captured only those design smells that consistently reappear in the literature. Finally, we have omitted ill-defined smells such as "unnamed coupling" [WCKD11] which lack an explicit definition but instead provide examples to describe the smell.

In this catalog, smells that violate a design principle are grouped together under that principle (see Figure 2). Smells in our catalog are named using our naming scheme and described using the template presented in Table 2; however, for the sake of brevity, we use a condensed version of this longer template here. The description of the smells in this section refer to "short description" element in the longer template given in Table 2.

In the following four subsections, we present design smells that violate the principles of *abstraction*, *encapsulation*, *modularity*, and *hierarchy* respectively.

5.1 Design Smells Violating the Principle of Abstraction

In this section, we use the term "abstraction" to limit the discussion to class level abstractions.

5.1.1 Incomplete Abstraction

This smell arises when an abstraction does not support a responsibility completely. This smell covers two possibilities:

- A responsibility is partially fulfilled by an abstraction, and the rest of it is not covered elsewhere.

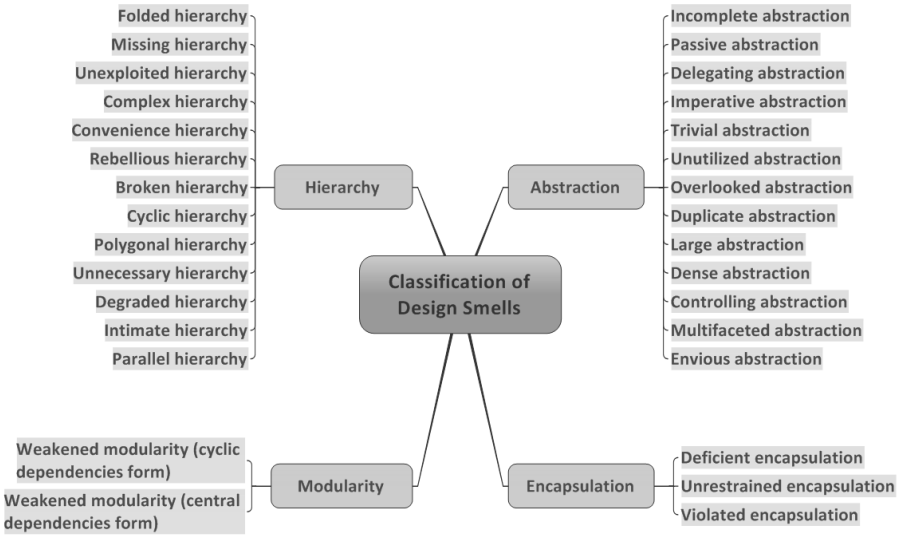

Figure 2 – A Principle-based Classification of Structural Design Smells

- A responsibility is split among two or more abstractions, resulting in incomplete abstraction(s).

Also known as: "Half-hearted operations" [SSM06], "class supports incomplete behavior" [PJ99].
Variants: "Inconsistent operations" [SSM06], "solution sprawl" [Ker04].

5.1.2 Passive Abstraction

This smell arises when a class is used as a holder for data, without any methods operating on it.

Also known as: "Data class" [CU06, Mar02], "record [class]" [GM05], "C-like structs" [Blo08], "no-command classes" [Mey00].
Variants: "Data clumps" [Fow99], "Cobol like [classes]" [GM05], "data container" [RFG05].

5.1.3 Delegating Abstraction

This design smell arises when an abstraction exists only for passing messages from one abstraction to another.

Also known as: "Agent classes" [LP09].
Variants: "Improper use of delegation" [CU06], "middle man" [Fow99], "using aggregation instead of inheritance" [Mil99].

5.1.4 Imperative Abstraction

This smell arises when an operation is turned into a class abstraction. This smell is evident when you encounter concrete classes with (a) the class name containing only a verb (or start with a verb), (b) having only one method with mostly the same name as the class, (c) the class shares no inheritance relationships.

Also known as: "Operation class" [LP09, RBP+91], "method turned into class" [CU06], "single-routine classes" [Mey00].

5.1.5 Trivial Abstraction

This design smell arises when a class is trivial and does not qualify to be one. This smell typically manifests itself as an abstraction with few (or no) methods or attributes, with no clear responsibility that could be assigned to it.

Also known as: "Irrelevant class" [LP09], "lazy class" [KPGA09, Fow99], "freeloader" [ZH11], "small class" [CU06, JR92], "mini-class" [SSM06], "not complex [class]" [Kho10].

Variants: "No responsibility" [Bud01].

5.1.6 Unutilized Abstraction

This smell arises when an abstraction is left unused (either not directly used or not reachable). This smell manifests in two forms:

Unreferenced abstractions: Concrete classes that are not being used by anyone.

Orphan abstractions: Stand-alone interfaces/abstract classes that do not have any subtypes.

Also known as: "Classes with unused responsibility" [Bud01], "hidden classes" [BBD+09], "ignored abstraction" [SSM06], "obsolete classes" [LR06], "unused classes" [LR06].

Variants: "Unused or little used items" [Gar07], "unnecessary design abstractions" [Sta07].

5.1.7 Overlooked Abstraction

This smell arises when clients refer to concrete types instead of their abstract base types. Two variants of this design smell are:

- The referred concrete types do not have relevant abstract base types defined, and the clients refer to the concrete types.
- The concrete types have abstract base types defined; however clients still refer to concrete derived types.

Also known as: "Interface bypass" [SSM06], "short-circuited abstraction" [TM05].

Variants: "Abstraction without decoupling" [DMTS10b, DMTS10a], "hidden relative" [SSM06].

5.1.8 Duplicate Abstraction

This design smell arises when there exist two or more abstractions that are similar (indicating that they share commonalities that have not been captured and utilized appropriately in the design). The design smell arises in many forms:

Identical name: The name of the abstractions are same.

Identical public interface: The abstractions have methods with same signature in its public interface.

Identical implementation: Logically the abstractions have similar implementation.

Also known as: "Similar signature class" [SSM06], "split identity" [SSM06], "alternative classes with different interfaces" [Fow99], "similar unrelated abstractions" [SSM06], "redundant classes" [RBP+91].

Variants: "Duplicate design artifacts" [Sta07], "similar classes" [BNL05].

5.1.9 Large Abstraction

This design smell arises when an abstraction has a large number of members in its public interface, its implementation, or both.

Also known as: "God class" [SSM06], "large class" [CU06, Mey00, Fow99, MGDM10].

Variants: "Higher relation" [CU06], "complex interface" [CU06], "large interface" [Rie96, CU06], "fat service with integrated interfaces" [Mar03], "big class" [Kre05], "blob" [BMB+98].

5.1.10 Dense Abstraction

This design smell arises when the abstraction has excessive implementation complexity.

Also known as: "Higher class complexity" [CU06].

Variants: "Too much responsibility" [Bud01], "module mimic" [MGDM10], "Spaghetti code" [BMB+98].

5.1.11 Controlling Abstraction

This smell arises when an abstraction controls other abstractions in the system.

Also known as: "Brain class" [LM06], "controller class" [Kho10, MGDM10].

Variant: "Abusive centralization of control" [TM05].

5.1.12 Multifaceted Abstraction

This design smell arises when an abstraction has more than one responsibility assigned to it.

Also known as: "Schizophrenic class" [TM05], "divergent change" [Fow99], "unconnected responsibilities" [Are04], "conceptualization abuse" [TM05], "mixed abstractions" [Mey00].

Variants: "Vague classes" [RBP+91], "blob" [BMB+98], "abusive conceptualization" [TM05], "Swiss army knife" [BMB+98], "overloaded services class" [Coa91], "non-related data and behavior" [CU06], "irrelevant methods" [CU06], "discordant attributes" [RBP+91].

5.1.13 Envious Abstraction

This design smell arises when methods in an abstraction are more interested in members of another abstraction (indicating misplacement of members).

Also known as: "Misplaced operations" [DDN02], "method in wrong class" [CU06], "feature envy" [Fow99].

Variants: "Misplaced control" [TM05], "message chains" [Fow99].

5.2 Design Smells Violating the Principle of Encapsulation

5.2.1 Deficient Encapsulation

This design smell arises when the encapsulation of an abstraction is insufficient. There are two degrees of deficiencies in encapsulation:

Lenient encapsulation: An encapsulation is lenient when the declared accessibility of one or more members of an abstraction is more permissive than actually required.

Vulnerable encapsulation: It occurs in the case when implementation details of an abstraction are exposed (or implementation details are inadequately protected) making it vulnerable to misuse or corruption of its state.

Also known as: "Public attributes" [Rie96, CU06], "poor encapsulated data" [CU06], "indecent exposure" [Ker04], "unhidden private method" [CU06], "encapsulation violation" [SSM06], "exposition of auxiliary method" [CWZ00], "not enough information hiding" [MGDM10], "class interface supports illegal or inappropriate states" [PJ99], "class interface supports illegal or dangerous behavior" [PJ99], "unrelated abstraction" [CU06].

Variants: "Weakening of data hiding" [CU06], "exposition of the internal structure of a collection" [CWZ00], "attribute visible to subclasses" [CWZ00].

5.2.2 Unrestrained Encapsulation

This design smell arises when an abstraction depends on the global state (global variables, data-structures etc.).

Also known as: "Unencapsulated class" [CU06], "class with unparameterized methods" [CU06].

Variants: "Higher external variables accesses" [CU06].

5.2.3 Violated Encapsulation

This design smell arises when a class-level abstraction directly accesses implementation details of other abstractions (which should have been ideally made inaccessible).

Also known as: "Violation of encapsulation" [DDN02], "inappropriate intimacy (general form)" [Fow99], "direct modification" [Bud01].

Variants: "Granting long-distance friendship" [Lak96], "access to foreign data" (metric) [LM06], "spaghetti scoping" [Szy92].

5.3 Design Smells Violating the Principle of Modularity

5.3.1 Weakened Modularity (Cyclic Dependencies Form)

This design smell arises when two or more class-level abstractions depend on each other directly or indirectly (creating a tight coupling among the abstractions).

Also known as: "Cyclic dependency between classes" [SSM06], "dependency cycles" [Sta07], "cyclic dependencies" [RW05], "cycles" [BNL05], "circular dependencies" [MGDM10], "static cycles in dependency graphs" [LR06], "bidirectional relation" [CU06], "cyclic class relationships" [Mil99].

Variants: "Group of interdependent objects" [MGDM10].

5.3.2 Weakened Modularity (Central Dependencies Form)

This design smell arises when a class-level abstraction has dependencies with large number of other class-level abstractions (high afferent and efferent coupling).

Also known as: "Bottlenecks" [RW05], "high collaboration class" [CU06], "class sends too many messages" [CU06], "god class" [DDN02, Rie96], "dispersed coupling" [LM06].

Variants: "Class collaborates with too many others" [Gar07], "man-in-the-middle" [RFG05], "intensive coupling" [LM06], "alien spider" [KBT07].

5.4 Design Smells Violating the Principle of Hierarchy

5.4.1 Folded Hierarchy

This design smell arises when there are abstractions which are amenable to more generalization in a hierarchy. This design smell manifests in two ways:

- When a suitable supertype could be identified from existing abstraction(s), but is currently missing [Gar07, DDN02]. Introducing supertype(s) can improve the hierarchy or result in forming a hierarchy.
- When data, behavior and/or interface is duplicated in abstractions that could be factored out to a suitable supertype [Rie96].

Also known as: "Orphan sibling method/attribute" [SSM06], "collapsed method hierarchy" [TM05], "repeated functionality" [Are04], "redundant variable declaration" [CU06], "missing abstract class" [Gar07], "missing levels of abstraction" [Mil99].

Variants: "Coarse hierarchies" [Mil99], "getting away from abstraction" [CU06], "generation conflict" [SSM06].

5.4.2 Missing Hierarchy

This design smell arises when an abstraction uses conditional logic on embedded features to determine behavior (such "tagging" indicates an unexploited is-a relationship).

Also known as: "Tag class" [Blo08], "missing inheritance" [DDN02], "collapsed type hierarchy" [TM05], "embedded features" [TM05].

5.4.3 Unexploited Hierarchy

This design smell arises when an abstraction has explicit condition checks instead of leveraging the polymorphism inherent in the hierarchy [TM05]. This includes the case where explicit type checks are performed.

Also known as: "Simulated polymorphism" [SSM06], "improper use of switch statement" [Fow99], "case analysis" [JF88], "instanceof checks" [EM02], "self type checks" [DDN02], "conditionals" [DDN02], "type queries" [LR06].

5.4.4 Complex Hierarchy

This design smell arises when the inheritance graph is tangled, or excessively wide, deep, or skewed [BBM96, Bin99, LR06].

Also known as: "Spaghetti inheritance" [JR92, Bin99], "complicated inheritance graphs" [JR92], "hierarchy of classes has too many levels" [Gar07], "list-like inheritance hierarchy" [LR06], "deep inheritance hierarchies" [Bin99, LR06], "large inheritance lattice" [BBM96], "huge inheritance/aggregation hierarchies" [Eic03], "monolithic hierarchies" [Dew02].

Variants: "Cosmic hierarchies" [Dew02].

5.4.5 Convenience Hierarchy

This design smell arises when the supertypes provide many concrete methods and subtypes do not override those concrete methods. This indicates use of inheritance primarily for code reuse deviating from the widely-accepted norm of using inheritance to model is-a relationships [Bin99, Sak89].

Also known as: "Abused inheritance" [TM05], "misuse of inheritance" [DDN02], "convenience inheritance" [Mey00], "haphazard inheritance" [AM94], "subclasses do not redefine methods" [LR06].

Variants: "Weird hierarchies" [Bin99], "inheritance for implementation reuse" [TM05], "incidental inheritance" [Sak89], "rare overriding" [Kho10].

5.4.6 Rebellious Hierarchy

This design smell arises when a subtype rejects or invalidates the methods supported from its supertype(s) [Gar07, Rie96]. In this smell, the supertype and subtype(s) conceptually share is-a relationship, but some methods break this relationship. Two forms of this design smell are:

Rebellious interface: The subtype does not support the interface of the supertype.

Rebellious implementation: The implementation of the operations in the subtype violates the contract of the supertype.

Also known as: "Harmful mutation" [HM95], "illoyalty of subclasses" [HM95], "refused bequest" [Fow99, Wak03], "refused interface" [TM05], "naughty children" [Bin99], "worm holes" [Bin99], "gnarly hierarchies" [Bin99], "premature interface abstraction" [TM05].

Variants: "Fat interface" [Bin99], "improper use of inheritance" [CU06], "misdeclared members of base class" [CU06].

5.4.7 Broken Hierarchy

This design smell arises when the supertype and its subtype(s) conceptually do not share "is-a" relationship (resulting in broken substitutability).

Also known as: "Inappropriate use of inheritance" [Bud01], "subclass is not true subtype of supertype" [PJ99], ""has" relation with no "is" relation" [Mey00], "containment by inheritance" [TM05].

Variants: "Mistaken aggregates" [PJ99], "misapplying is a" [PJ99], "object or variable turned into a subclass" [CU06], "improper inheritance" [Mil99], "subclass inheriting inappropriate operations from supertype" [PJ99], "cancelled local behavior but supertype reuse" [ADGN10], "tradition breaker" [LM06], "inverted hierarchies" [PJ99], "inverse abstraction hierarchies" [Mil99].

5.4.8 Cyclic Hierarchy

This design smell arises when a supertype directly or indirectly refers to one of its subtype, forming a cycle in the hierarchy.

Also known as: "Knows of derived" [Ciu99, SSM06], "subtype knowledge" [DMTS10b], "subclass knowledge" [BNL05], "curious superclasses" [BNL05], "inheritance/reference cycles" [SSC96], "base class depends on derived classes" [CWZ00], "forward associations in a hierarchy" [Mil99].

Variants: "Inheritance loops" [Bin99].

5.4.9 Polygonal Hierarchy

This design smell arises when a supertype is repeatedly inherited in descendant abstraction(s) (forming polygons in the inheritance graph). **Also known as:** "Rhombus-like inheritance" [BLS00], "diamond-shaped inheritance" [Str00], "diamond problem" [TJJV04], "degenerated inheritance" [DMTS10b], "fork-join inheritance" [Sak89], "repeated inheritance" [Mey00].

Variants: "Common ancestor dilemma" [TJJV04], "multiple inheritance use" [Rie96], "poor usage of interfaces" [MGDM10], "inheritance loops" [Bin99].

5.4.10 Unnecessary Hierarchy

This design smell arises when a hierarchy has one or more unnecessary abstractions. This smell includes the case when a supertype has at most one concrete subtype (indicating that generalization was based on imagination rather than on concrete specializations that can be generalized).

Also known as: "Extra sub-class" [CU06], "taxomania" [Mey00], "speculative generality" [Fow99].

Variants: "Inheritance hierarchies without polymorphic assignments" [LR06], "lazy base class" [CU06].

5.4.11 Degraded Hierarchy

This design smell arises when the hierarchy tends to be more concrete towards the root and more abstract towards the leaves. This smell includes the case where a supertype is declared concrete and its subtype is declared abstract.

Also known as: "Illegal abstract inheritance" [Ber04], "super class is a concrete class" [Gar07], "illegal abstract inheritance" [Ber04], "abstract leaf classes" [MD01].

Variants: "Abstracting concrete methods" [ADGN10], "inflexible root class" [Mil99], "inverse abstraction hierarchies" [Mil99].

5.4.12 Intimate Hierarchy

This design smell arises when the subtype(s) directly access the state of supertype(s) [ADGN10]. **Also known as:** "Inappropriate intimacy (subclass form)" [Wak03, Fow99], "ancestor direct state access" [ADGN10].

5.4.13 Parallel Hierarchy

This design smell arises when there are two structurally similar (symmetrical) class hierarchies with same class name prefixes [Fow99].

Also known as: "Parallel inheritance hierarchies" [Fow99, LR06].

6 Initial Evaluation

In order to evaluate the usefulness of our design smells catalog, classification, and naming scheme, as an initial step, we created a design smells-centric homework assignment for an intensive software design training at Siemens CT DC AA [CTD]. All of the 12 training participants were software architects with an average experience of 10.8 years in the software industry and 2.9 years as an architect. The assignment was 1.5 months in duration and required these architects to use the catalog as a reference to find and report design smells in their current and past projects.

Q. No	Question [note in square brackets]	Responses: Mean/Summary
1.	Rate your knowledge in design smells before this assignment on design smells. [0 - no knowledge; 5 - "expert" level knowledge]	1.9
2.	Rate your knowledge in design smells after this assignment on design smells. [0 - no knowledge; 5 - "expert" level knowledge]	3.7
3.	How useful was the design smells catalog to understand the kind and extent of design problems in your project? [0 - the catalog was of no use; 5 - the catalog was extremely useful.]	4.0
4.	Have you come across any design smells you've never seen before but was provided in this catalog? [Yes/No] If yes, please list them.	Yes - 9; No - 2; No response - 1
5.	Did you find any design smells in your project that were not covered in this catalog? [Yes / No.] If yes, please list them.	Yes - 3; No - 6; No response - 3
6.	Rate the extent to which you believe that the smells found in your project (during the assignment) actually resulted from a violation of the corresponding design principles. [0 - none of the design smells resulted from violation of the corresponding design principle; 5 - all of the design smells resulted from violation of the design principle]	3.7
7.	How useful is the naming scheme in the design smells catalog to understand the cause of design smell? [0 - not at all understandable; 5- very easy to understand]	3.8
8.	How intuitive is the naming scheme in the design smells catalog? [0 - not at all intuitive; 5 - very intuitive]	3.9
9.	Would you use the design smell catalog to identify smells in the future for your project(s)? [Yes / No]	Yes - 11; No - 1
10.	Would you recommend this design smell catalog to be used by architects in other projects that you know? [Yes / No.]	Yes - 11; No - 1

Table 3 – Feedback from architects on design smell catalog

Upon the completion of the assignment, we used a questionnaire to elicit feedback from these architects on their experience with using our design smells catalog. The questionnaire consisted of 10 rating-based questions, with an option to enter additional comments. We received responses from all the 12 participants. Table 3 lists the questions and the quantified responses from the participants.

We now list some noteworthy aspects that emerged from the feedback we received.

On the catalog: The fact that the increase in design smells knowledge is close to a factor of two (see questions 1 & 2) indicates that our catalog can serve as a central repository of design smells knowledge for practitioners. In fact, one of the participants mentioned that the catalog was useful as a quick reference for identifying smells. This is further supported by the responses to questions 3 and 4. Specifically, the participants found the design smells catalog "very useful" to understand the kind and extent of design problems in their projects. 75% of the participants also reported that they became aware of a few new smells.

In response to question 5, new smells were reported by 3 participants. A careful analysis of these smells revealed that they were either architectural smells (e.g. problems with layering) or behavioral smells (e.g. performance related). Since these smells are outside the scope of our current work (see Table 1), we do not discuss them in detail here. The response to this question suggests that from a practitioner's perspective, the catalog is reasonably comprehensive.

On the principle-based classification scheme: The architects believe that "most of the smells" (3.7 in the scale of 0 to 5; see question 6) they found in their projects actually resulted from a violation of the corresponding design principle. An analysis of the responses with low-rating reveal that the architects instead considered factors such as time pressures and contextual constraints to lead to design smells. However, a deeper reflection on this issue reveals that the aforementioned factors in fact eventually lead to the violation of underlying design principles that manifest as smells.

One of the respondent mentioned that, "I know about design smells as well as design principles, but this was the first time I could see violation of these principles documented as smells." This lends credence to the usefulness of the principle-based classification scheme presented in this work.

On the naming scheme: Most of the architects found the naming scheme useful as well as intuitive (questions 7 & 8). One participant mentioned that he liked names like "envious abstraction" which helped him to understand the cause of the design smell.

On the experience of using the catalog: Of the 12 architects, 11 were affirmative on using the design smell catalog to identify smells in the future in their projects; further, they were willing to recommend this design smell catalog to be used in other projects as well (questions 9 and 10). One architect added: "Generally I use my experience to identify the design smells. But the catalog provided in the assignment gave me a good reference about the different types of design smells that can exist such that identifying the design smells becomes much easy." Responses to these two questions clearly show the positive light in which this design smells work has been received by architects in Siemens CT DC AA.

In summary, the overall feedback indicates that all the three aspects of our design smells work - the catalog, the classification, as well as the naming scheme - has been received positively by the practitioners. However, our evaluation suffers from two limitations. First, this is an initial evaluation with feedback received only from 12 participants. Second, focus of the evaluation was limited to our design smells work and comparing our work against others requires more elaborate evaluation. We plan to address both these limitations in future with more extensive evaluations.

7 Discussion

In this section, we summarize our efforts and present some interesting observations that we made while cataloging and classifying design smells. We believe insights resulting from these observations point towards potential future advances in the area of software design smells. They also provide motivation for the development of methodology and tools that can provide better guidance to practitioners during the design process.

7.1 Enhancing the catalog with more smells

While collecting and cataloging smells, we noticed that there are smells that are commonly found in industrial designs but have not yet been well-documented in literature. A deeper reflection on why some smells have remained undocumented or not so well-documented seems to indicate that the community has not focused much on the actual cause of smells itself. We believe that a larger exploration of the smell space is possible if one were to investigate how the violation of a principle leads to a smell. However, there is little work done in exploring the bridge between a design principle and the smell that is caused when the principle is violated. An insight that results from this is that a systematic consideration of techniques that enable the realization of design principles is needed in order to properly make the connection between a principle and a smell. Once these techniques are identified, the possible occurrence of a smell can be predicted whenever those techniques are not followed.

For instance, consider the criteria of sufficiency, completeness, and primitiveness which an abstraction should satisfy as per Booch's proposition [BME+07]. While smells related to the completeness criteria (referred under Section 5.1.1 "incomplete abstraction" in our catalog) are well-documented, smells related to the criteria of sufficiency and primitiveness are not well-documented, as already pointed out above. Other examples include the techniques of "separation of interface from implementation" [GHJV95] which can be used for encapsulation and "separate policy from implementation" [BMR+96]. Design smells that occur as a result of not following these techniques are not well documented in literature. If we were to use these "enabling techniques" as a means of finding smells, it would be possible to identify or discover new smells. For instance, if we were to use sufficiency and primitiveness as enabling techniques for abstraction, we would be able to identify new smells that could be termed "insufficient abstraction" and "non-primitive abstraction" respectively. We believe this insight leads to the idea that enabling techniques can play an important role in the future classification of smell, and even identification of new smells.

It should be pointed out that there are some instances of explicit mapping between smells and enabling techniques. For example, inFusion tool has a smell named "SAP breaker" [InF12] which breaks Stable Abstractions Principle (SAP) [Mar03]. However, using such enabling techniques to bridge a connection between principles and smells has largely been unexplored.

7.2 Consistency of Names of Principles

While cataloging smells, we found it difficult to name design smells that occur because of the violation of the principle of "modularity". This is partly because our naming scheme needed to adhere to the constraints listed in Section 7.3 and partly because "modularity" is inherently defined as a property of the entire system (or architecture). Hence, modularity was more difficult to qualify compared to the other principles. We have explored some ways that could potentially address the naming issue with modularity. For instance, one option could be to use the term "modularization" instead of "modularity". While this would definitely help ease the task of naming smells under modularity, on the downside we would be deviating from the standard principles as named by Booch and which the community has come to accept. Another option could be to adopt a completely new naming scheme. However, our initial experiments with other naming schemes have indicated that no scheme is ideal and as mentioned above, a naming scheme that meets all the constraints listed in Section 7.3 will require

sustained and ongoing effort.

7.3 Improved Naming Scheme for Smells

Our survey of the existing literature showed that there is a lack of a common consistent vocabulary for design smells. To address this issue and to specifically also provide a naming scheme for smells which is meaningful, we introduced a simple and uniform naming scheme for design smells based on our classification framework. Under our current naming approach, the name of each design smell is not only short and therefore easy to remember but also indicates clearly how one of the four design principles has been violated.

While this naming scheme appears to have served reasonably well for the purpose of uniform naming of design smells, we believe that realizing an ideal naming scheme is an ongoing effort. This is mainly because there are several constraints impacting the naming scheme. In particular, we want a naming scheme that not only helps map smells to principles (which is why a principle is mentioned in each smell name) but one that is also be simple and easy to remember (which constrains the number of terms in the smell name) as well as uniform and consistent (which implies all smells should be named in a similar fashion). Managing these constraints can result in certain unintuitive names for smells. For example, just by reading the name "unrestrained encapsulation" (Section 5.2.2), it is not evident that the smell arises when an abstraction depends on the global state. Sometimes, the difference between the smells is not evident from the names itself. For example, it is difficult to distinguish between "missing hierarchy" (Section 5.4.2) and "unexploited hierarchy" (Section 5.4.3). Based on their names, both smells indicate a deviation from the principle of hierarchy and the real difference is evident only after reading the descriptions of these smells.

8 Future Directions

The work we have presented in this paper is our initial attempt at cataloging, classifying, and naming structural design smells. Based on the observations and insights described in the previous sections, we envision that our work can be extended in future along the following dimensions.

- Augment our catalog with smells that have been undocumented in literature.
- Broaden the current classification scheme by including more design principles so that more smells can be cataloged.
- Introduce enabling techniques into the current classification scheme to enable a more effective mapping between design principles and smells.
- Explore how smells can be described in a semi-formal or formal notation in order to address the current ambiguity in interpreting design smells documented in literature.
- Leverage our experience with structural micro-architectural smells to similarly collect and categorize architectural smells.
- Expand the smell template to include key information in the description of smells in our catalog, for example, with detailed refactoring steps that could

help address that smell (the design principles only provide high-level refactoring hints).

- Explore the relationship between the smells (such as how one smell "can cause", "typically occur with", or "lead to" other smell(s)).
- Develop tool support for the automatic detection of our cataloged smells in real-world software. Tools that analyze software and identify our cataloged smells would be of immense benefit to practitioners; our survey of literature and design analysis tools, however, shows that there is a lack of comprehensive tools that can detect all the design smells in our catalog. Further, such tool support can also be leveraged to conduct empirical studies to help answer questions such as which smells are more frequently found in real-world software, which smells tend to occur together, and how useful is our classification and naming scheme.
- As mentioned earlier (see Section 6), we plan to undertake a comprehensive evaluation activity to compare the effectiveness and usefulness of our design smell classification scheme with the other classification schemes.

References

[ADGN10] G. Arévalo, S. Ducasse, S. Gordillo, and O. Nierstrasz. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Inf. Softw. Technol.*, 52(11):1167–1187, November 2010. URL: `http://dx.doi.org/10.1016/j.infsof.2010.05.010`, `doi:10.1016/j.infsof.2010.05.010`.

[AM94] J. M. Armstrong and R. J. Mitchell. Uses and abuses of inheritance. *Software Engineering Journal*, 9(1):19 –26, jan 1994.

[Are04] G. Arevalo. High-level views in object-oriented systems using formal concept analysis. *PhD Thesis, The University of Bern,*, 2004.

[BBD+09] F. Balmas, A. Bergel, S. Denier, S. Ducasse, J. Laval, K. Mordal-Manet, H. Abdeen, and F. Bellingard. Software metric for java and c++ practices. Technical report, Squale Project, 2009.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, October 1996. URL: `http://dx.doi.org/10.1109/32.544352`, `doi:10.1109/32.544352`.

[BD02] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, January 2002. URL: `http://dx.doi.org/10.1109/32.979986`, `doi:10.1109/32.979986`.

[Ber04] B. Berenbach. The evaluation of large, complex uml analysis and design models. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 232–241, Washington, DC, USA, 2004. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=998675.999428`.

[Bin99] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Blo08] J. Bloch. *Effective Java*. Addison-Wesley; 2 edition, 2008.

[BLS00] D. Beyer, C. Lewerentz, and F. Simon. Flattening inheritance structures – or – Getting the right picture of large OO-systems. Technical report, Institute of Computer Science, Brandenburgische Technische Universität Cottbus, November 2000.

[BMB+98] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[BME+07] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston. *Object-oriented analysis and design with applications (third edition)*. Addison-Wesley Professional, 2007.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[BNL05] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Softw. Eng.*, 31(2):137–149, February 2005. URL: `http://dx.doi.org/10.1109/TSE.2005.23`, `doi:10.1109/TSE.2005.23`.

[Bud01] T. Budd. *An introduction to object-oriented programming*. Addison Wesley; 3 edition, 2001.

[Ciu99] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '99, pages 18–, Washington, DC, USA, 1999. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=832257.833062`.

[Coa91] P. Coad. Ood criteria, part 3. *Journal of Object Oriented Programming*, September, 1991.

[CTD] CT DC AA homepage: `www.siemens.co.in/en/about_us/index/innovations/CTDCIN.htm`.

[CU06] M. Choinzon and Y. Ueda. Detecting defects in object oriented designs using design metrics. In *Proceedings of the 2006 conference on Knowledge-Based Software Engineering: Proceedings of the Seventh Joint Conference on Knowledge-Based Software Engineering*, pages 61–72, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press. URL: `http://dl.acm.org/citation.cfm?id=1565098.1565107`.

[Cun92] W. Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992. URL: `http://doi.acm.org/10.1145/157710.157715`, `doi:10.1145/157710.157715`.

[CWZ00] A. L. Correa, C. M. L. Werner, and G. Zaverucha. Object oriented design expertise reuse: An approach based on heuristics, design patterns and anti-patterns. In *Proceedings of the 6th International Conerence on Software Reuse: Advances in Software Reusability*, ICSR-

6, pages 336–352, London, UK, UK, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645546.656055.

[DDN02] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Dew02] S. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[DMTS10a] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah. Barriers to modularity: an empirical study to assess the potential for modularisation of java programs. In *Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice - Reality and Gaps*, QoSA'10, pages 135–150, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-13821-8_11, doi:10.1007/978-3-642-13821-8_11.

[DMTS10b] J. Dietrich, C. McCartin, E. D. Tempero, and S. M. A. Shah. On the detection of high-impact refactoring opportunities in programs. *CoRR*, abs/1006.1747, 2010.

[Eic03] H. Eichelberger. Nice class diagrams admit good design? In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 159–ff, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/774833.774857, doi:10.1145/774833.774857.

[EM02] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 97–, Washington, DC, USA, 2002. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=882506.885134.

[Fow99] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Gar07] J. Garzas. *Object-Oriented Design Knowledge: Principles, Heuristics, and Best Practices*. IGI Publishing, Hershey, PA, USA, 2007.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GM05] J. Y. Gil and I. Maman. Micro patterns in java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 97–116, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1094811.1094819, doi:10.1145/1094811.1094819.

[HKGH10] S. Hassaine, F. Khomh, Y. Gueheneuc, and S. Hamel. Ids: An immune-inspired approach for the detection of software design smells. In *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, QUATIC '10, pages 343–348, Washington, DC, USA, 2010. IEEE Computer Society. URL: http://dx.doi.org/10.1109/QUATIC.2010.61, doi:10.1109/QUATIC.2010.61.

[HM95] M. Hitz and B. Montazeri. Measuring product attributes of object-oriented systems. In *Proceedings of the 5th European Software Engineering Conference*, pages 124–136, London, UK, UK, 1995. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645385.651487.

[InF12] Infusion hydrogen: Design flaw detection tool. Available at: http://www.intooitus.com/products/infusion, 2012.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1:22–35, June/July 1988.

[JR92] P. Johnson and C. Rees. Reusability through fine-grain inheritance. *Softw. Pract. Exper.*, 22(12):1049–1068, December 1992. URL: http://dx.doi.org/10.1002/spe.4380221203, doi:10.1002/spe.4380221203.

[KBT07] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparol and a software evolution ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society. URL: http://dx.doi.org/10.1109/MSR.2007.21, doi:10.1109/MSR.2007.21.

[Ker04] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[Kho10] F. Khomh. Patterns and quality of object-oriented software systems. *PhD Thesis, University of Montreal*, 2010.

[KPGA09] F. Khomh, M. D. Penta, Y. Guehéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on software changeability. *Technical report EPM-RT-2009-02, ecole Polytechnique de Montreal*, April 2009.

[Kre05] J. Kreimer. Adaptive detection of design flaws. *Electron. Notes Theor. Comput. Sci.*, 141(4):117–136, December 2005. URL: http://dx.doi.org/10.1016/j.entcs.2005.02.059, doi:10.1016/j.entcs.2005.02.059.

[Lak96] J. Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

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

[LP09] M. T. Llano and R. Pooley. Uml specification and correction of object-oriented anti-patterns. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 39–44, Washington, DC, USA, 2009. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ICSEA.2009.15, doi:10.1109/ICSEA.2009.15.

[LR06] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[LSP05] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 214–223, New York, NY, USA,

2005. ACM. URL: http://doi.acm.org/10.1145/1101908.1101941, doi:10.1145/1101908.1101941.

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

[Mar03] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Addison-Wesley, 2003.

[MD01] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 83–86, New York, NY, USA, 2001. ACM. URL: http://doi.acm.org/10.1145/602461.602476, doi:10.1145/602461.602476.

[Mey00] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[MGDM10] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, January 2010. URL: http://dx.doi.org/10.1109/TSE.2009.50, doi:10.1109/TSE.2009.50.

[Mil99] B. K. Miller. Object-oriented architecture measures. In *Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8 - Volume 8*, HICSS '99, pages 8069–, Washington, DC, USA, 1999. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=874074.876299.

[MRB97] R. C. Martin, D. Riehle, and F. Buschmann, editors. *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[MVL03] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 381–, Washington, DC, USA, 2003. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=942800.943571.

[PJ99] M. Page-Jones. *Fundamentals of object-oriented design in UML*. Addison-Wesley Professional; 1 edition, 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[RFG05] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1082983.1083155, doi:10.1145/1082983.1083155.

[Rie96] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Professional, 1996.

[RW05] R. Ramler and K. Wolfmaier. Common findings and lessons learned from software architecture and design analysis. *Proc. 11th IEEE Int. Software Metrics Symposium*, September 2005.

[Sak89] M. Sakkinen. Disciplined Inheritance. *ECOOP'89: Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 39–56, 1989.

[SRK07] S. Sarkar, G. M. Rama, and A. C. Kak. Api-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. Softw. Eng.*, 33(1):14–32, January 2007. URL: `http://dx.doi.org/10.1109/TSE.2007.4`, `doi:10.1109/TSE.2007.4`.

[SSC96] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=227726.227810`.

[SSM06] F. Simon, O. Seng, and T. Mohaupt. *Code Quality Management: Technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht*. dpunkt-Verlag, 2006.

[Sta07] M. Stal. Software architecture refactoring. *Tutorial, in The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[Szy92] C. A. Szyperski. Import is not inheritance - why we need both: Modules and classes. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '92, pages 19–32, London, UK, UK, 1992. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=646150.679345`.

[TJJV04] E. Truyen, W. Joosen, B. N. Jorgensen, and P. Verbaeten. A generalization and solution to the common ancestor dilemma problem in delegation-based object systems. *Proceedings of the 2004 Dynamic Aspects Workshop*, 2004.

[TM05] A. Trifu and R. Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 155–164, Washington, DC, USA, 2005. IEEE Computer Society. URL: `http://dx.doi.org/10.1109/WCRE.2005.15`, `doi:10.1109/WCRE.2005.15`.

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

[WCKD11] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 411–420, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/1985793.1985850`, `doi:10.1145/1985793.1985850`.

[ZH11] D. Zaparanuks and M. Hauswirth. The beauty and the beast: separating design from algorithm. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 27–51, Berlin, Heidelberg, 2011. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=2032497.2032502`.

About the authors

S G Ganesh is a practitioner currently working in the area of code quality management in Siemens Corporate Research and Technologies in Bangalore. Prior to Siemens, he worked in HP's C++ compiler team for 5 years and was also a member of C++ standardization committee. His areas of interests include OO design, design patterns, and programming languages. He is a Software Engineering Certified Instructor (IEEE certification). Contact him at `ganesh.samarthyam@siemens.com`.

Tushar Sharma is a researcher and practitioner at Siemens Corporate Research & Technologies-India for the last five years. His research interests include OO software design, OO programming, refactoring, and design patterns. Contact him at `tushar.sharma@siemens.com`.

Girish Suryanarayana works as a researcher and a consultant architect at Siemens Corporate Research & Technologies India. He received a PhD in Information and Computer Science from the University of California, Irvine in 2007. His interests lie primarily in the area of software architecture. Girish is a IEEE-certified SECI (Software Engineering Certified Instructor) and regularly conducts trainings on software architecture and design. Contact him at `girish.suryanarayana@siemens.com`.

Acknowledgments We thank Prof. K. V. Dinesha (IIIT, Bangalore) and Christian Körner (Siemens Corporate Research and Technologies, Munich) for their suggestions during the early stages of this work.