

Design Smells

2

Do you smell it?

That smell.

A kind of smelly smell.

The smelly smell that smells...smelly

Mr. Krabs¹

The previous chapter introduced technical debt and its impact on a software organization. We also looked at the various factors that contribute to technical debt. One such factor is inadequate awareness of design smells and refactoring. We have repeatedly observed in software development projects that designers are aware of the fundamental design principles, but sorely lack knowledge about the correct application of those principles. As a result, their design becomes brittle and rigid and the project suffers due to accumulation of technical debt.

Psychologists have observed that mistakes are conducive to learning, and have suggested that the reason lies in the element of surprise upon finding out that we are wrong. There is also a well-known cliché that we learn more from our mistakes than our successes.

This book, therefore, adopts an approach different from the traditional approaches towards software design. Instead of focusing on design principles and applying them to examples, we take examples of design *smells*, and discuss why they occurred and what can be done to address them. In the process, we reveal the design principles that were violated or not applied properly. We believe that such an approach towards design that focuses on smells will help the readers gain a better understanding of software design principles and help them create more effective designs in practice.

Before we delve into the catalog of smells, we first discuss why we need to care about smells and various factors that lead to the occurrence of smells. We then present a simple, yet powerful classification framework that helps categorize smells based on the design principles that the smells violate.

¹ “SpongeBob SquarePants”, episode “Help Wanted”, season one, aired on May 1, 1999.

2.1 WHY CARE ABOUT SMELLS?

One of the key indicators of technical debt is poor software quality. Consider some of the common challenges that developers face while working in software projects that are accumulating technical debt:

- The software appears to be getting insanely complex and hard to comprehend. Why is the “understandability” of software getting worse?
- The software continues to change with requests for defect fixes and enhancements and takes increasingly more time for the same. Why is the software’s “changeability” and “extensibility” getting worse?
- Logically, there seem to be many aspects or parts of the software that can be reused. However, why is it becoming increasingly difficult to reuse parts of the software (i.e., why is its “reusability” getting worse)?
- Customers are becoming increasingly unhappy about the software and want a more reliable and stable product. The quality assurance team is finding it more difficult to write tests. Why is the software’s “reliability” and “testability” getting worse?

Since software design is known to have a major impact on software quality, and the focus of this book is on software design, we summarize these above qualities² in the context of software design in [Table 2.1](#). Note that we use the term *design fragment* in this table to denote a part of the design such as an abstraction (e.g., a class,

Table 2.1 Important Quality Attributes and Their Definitions

Quality Attribute	Definition
Understandability	The ease with which the design fragment can be comprehended.
Changeability	The ease with which a design fragment can be modified (without causing ripple effects) when an existing functionality is changed.
Extensibility	The ease with which a design fragment can be enhanced or extended (without ripple effects) for supporting new functionality.
Reusability	The ease with which a design fragment can be used in a problem context other than the one for which the design fragment was originally developed.
Testability	The ease with which a design fragment supports the detection of defects within it via testing.
Reliability	The extent to which the design fragment supports the correct realization of the functionality and helps guard against the introduction of runtime problems.

²Please note that there are numerous other important quality attributes (such as performance and security) that are impacted by structural design smells. However, we believe an in-depth discussion of these qualities and their relationship to design smells is an entire book in itself, and hence omitted it from this book.

interface, abstract class) or a collection of abstractions and their relationships (e.g., inheritance hierarchies, dependency graphs).

We need to care about smells because smells negatively impact software quality, and poor software quality in turn indicates technical debt. To give a specific example, consider a class that has multiple responsibilities assigned to it in a clear violation of the Single Responsibility Principle (see Appendix A). Note that this smell is named Multifaceted Abstraction in this book (see Section 3.4). Table 2.2 provides an

Table 2.2 Impact of Multifaceted Abstraction on Key Quality Attributes

Quality Attribute	Impact of Multifaceted Abstraction on the Quality Attribute
Understandability	A class with the Multifaceted Abstraction smell has multiple aspects realized into the abstraction, increasing the cognitive load on the user. When a class has multiple responsibilities, it takes more time and effort to understand each responsibility, how they relate to each other in the abstraction, etc. This adversely affects its understandability.
Changeability & extensibility	When a class has multiple responsibilities, it is difficult to determine which members should be modified to support a change or enhancement. Further, a modification to a member may impact unrelated responsibilities within the same class; this in turn can have a ripple effect across the entire design. For this reason, the amount of time and effort required to change or extend the class while still ensuring that the resulting ripple effect has no adverse impact on the correctness of the software is considerably greater. These factors negatively impact changeability and extensibility.
Reusability	Ideally, a well-formed abstraction that supports a single responsibility has the potential to be reused as a unit in a different context. When an abstraction has multiple responsibilities, the entire abstraction must be used even if only one of the responsibilities needs to be reused. In such a case, the presence of unnecessary responsibilities may become a costly overhead that must be addressed. Thus the abstraction's reusability is compromised. Further, in an abstraction with multiple responsibilities, sometimes the responsibilities may be intertwined. In such a case, even if only a single responsibility needs to be reused, the overall behavior of the abstraction may be unpredictable, again affecting its reusability.
Testability	Often, when a class has multiple responsibilities, these responsibilities may be tightly coupled to each other, making it difficult to test each responsibility separately. This can negatively impact the testability of the class.
Reliability	The effects of modification to a class with intertwined responsibilities may be unpredictable and lead to runtime problems. For instance, consider the case in which each responsibility operates on a separate set of variables. When these variables are put together in a single abstraction, it is easy to mistakenly access the wrong variable, resulting in a runtime problem.

overview of how this smell impacts the design quality in terms of the quality attributes defined in [Table 2.1](#). Note that the specific impact of each smell on key quality attributes is described in Chapters 3 to 6.

The impact of design smells is not limited to just software quality. In some cases, it can even severely impact the reputation of the organization. For instance, the presence of design smells in a framework or a library (i.e., software that exposes an Application Programming Interface(API)) that is going to be used by clients can adversely impact how the organization is perceived by the community. This is because it is hard to fix design smells in the API/framework once the clients start using the API.

2.2 WHAT CAUSES SMELLS?

Since smells have an impact on design quality, it is important to understand smells and how they are introduced into software design. We want to point out that since design smells contribute to technical debt, there is some overlap in the causes of design smells and technical debt. Thus, some of the causes of technical debt that were discussed in the previous chapter are relevant here, for example, lack of good designers and lack of refactoring; to avoid repetition, we did not include them here. [Figure 2.1](#) shows a pictorial summary of the causes of smells discussed in this section.

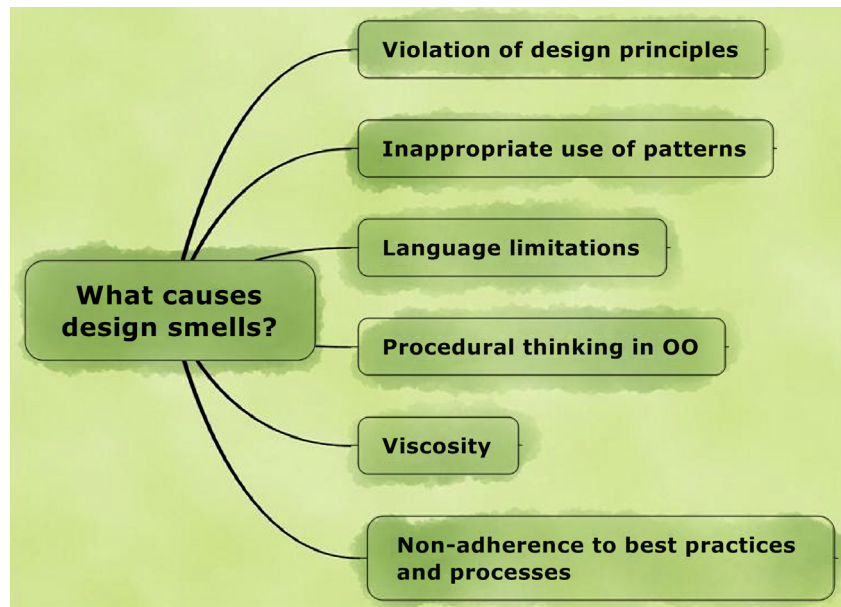


FIGURE 2.1

Common causes of design smells.

2.2.1 VIOLATION OF DESIGN PRINCIPLES

Design principles provide guidance to designers in creating effective and high-quality software solutions. When designers violate design principles in their design, the violations manifest as smells.

Consider the `Calendar` class that is part of the `java.util` package. A class abstracting real-world calendar functionality is expected to support date-related functionality (which it does), but the `java.util.Calendar` class supports time-related functionality as well. An abstraction should be assigned with a unique responsibility. Since `java.util.Calendar` class is overloaded with multiple responsibilities, it indicates the violation of the principle of abstraction (in particular, violation of the Single Responsibility Principle). We name this the Multifaceted Abstraction (see Section 3.4) smell because the class supports multiple responsibilities.

Similarly, consider the class `java.util.Stack` which extends `java.util.Vector`. `Stack` and `Vector` conceptually do not share an IS-A relationship because we cannot substitute a `Stack` object where an instance of `Vector` is expected. Hence this design indicates a violation of the principle of hierarchy (specifically, the violation of the principle of substitutability - see Appendix A). We name this smell Broken Hierarchy (Section 6.8) since substitutability is broken.

2.2.2 INAPPROPRIATE USE OF PATTERNS

Sometimes, architects and designers apply well-known solutions to a problem context without fully understanding the effects of those solutions. Often, these solutions are in the form of design patterns, and architects/designers feel pressured to apply these patterns to their problem context without fully understanding various forces that need to be balanced properly. This creates designs that suffer from symptoms such as too many classes or highly coupled classes with very few responsibilities [81].

Applying design patterns must be a very methodical and thought-out process. A design pattern, as captured by its class and sequence diagram notations, is only a reference solution; thus, there can be hundreds of variants of the design pattern, each with a particular consequence. An architect/designer who does not fully understand the finer aspects and implications of using a particular variation can end up severely impacting the design quality.

There is an interesting interplay between design smells and design patterns. Often, the most suitable way to address a design smell is to use a particular design pattern. However, it is also the case that the (wrong, unnecessary, or mis-)application of a design pattern can often lead to a design smell (also called an *antipattern*)!

2.2.3 LANGUAGE LIMITATIONS

Deficiencies in programming languages can lead to design smells. For example, Java did not support enumerations in its initial versions, hence programmers were forced to use classes or interfaces to hold constants. This resulted in the introduction of Unnecessary Abstraction smell (Section 3.5) in their designs. As another example, consider the

classes `AbstractQueuedSynchronizer` and `AbstractQueuedLongSynchronizer` from JDK. Both classes derive directly from `AbstractOwnableSynchronizer` and the methods differ in the primitive types they support (`int` and `long`). This resulted in an Unfactored Hierarchy smell (see Section 6.3). Since the generics feature in Java does not support primitive types, it is not possible to eliminate such code duplication when programming in Java.

2.2.4 PROCEDURAL THINKING IN OO

Often, when programmers with procedural programming background transition to object-oriented paradigm, they mistakenly think of classes as “doing” things instead of “being” things. This mindset manifests in the form of using imperative names for classes, functional decomposition, missing polymorphism with explicit type checks, etc., which result in design smells in an object-oriented context (for example, see Section 3.2).

2.2.5 VISCOSITY

One of the reasons that developers may resort to hacks and thus introduce design smells instead of adopting a systematic process to achieve a particular requirement is *viscosity* [79]. Viscosity is of two types: software viscosity and environment viscosity. Software viscosity refers to the “resistance” (i.e., increased effort and time) that must be encountered when the correct solution is being applied to a problem. If, on the other hand, a hack requires less time and effort (i.e., it offers “low resistance”), it is likely that developers will resort to that hack, giving rise to design smells.

Environment viscosity refers to the “resistance” offered by the software development environment that must be overcome to follow good practices. Often, if the development environment is slow and inefficient and requires more time and effort to follow good practices than bad practices, developers will resort to bad practices. Factors that contribute to the environment viscosity include the development process, reuse process, organizational requirements, and legal constraints.

2.2.6 NONADHERENCE TO BEST PRACTICES AND PROCESSES

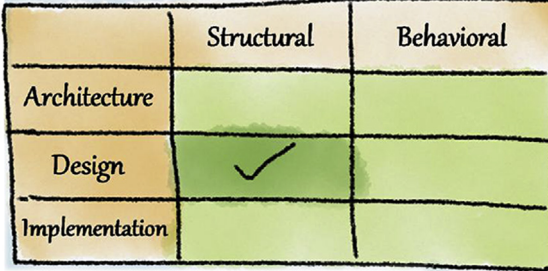
Industrial software development is a complex affair that involves building of large-scale software by a number of people over several years. One of the ways such complexity can be better managed is through adherence to processes and best practices. Often, when a process or practice is not followed correctly or completely, it can result in design smells. For instance, a best practice for refactoring is that a “composite refactoring” (i.e., a refactoring that consists of multiple steps [13]) should be performed atomically. In other words, either all or no steps in the refactoring should be executed. If this best practice is not adhered to and a composite refactoring is left half-way through, it can lead to a design smell (see anecdote in Section 6.9 for an example).

2.3 HOW TO ADDRESS SMELLS?

Clearly, smells significantly impact the design quality of a piece of software. It is therefore important to find, analyze, and address these design smells. Performing refactoring is the primary means of repaying technical debt. Note that the refactoring suggestions for each specific smell are described as part of the smell descriptions in Chapters 3 to 6. Further, Chapter 8 provides a systematic approach for repaying technical debt via refactoring.

2.4 WHAT SMELLS ARE COVERED IN THIS BOOK?

Smells can be classified as architectural, design (i.e., microarchitectural), or implementation level smells. It is also possible to view smells as structural or behavioral smells. We limit our focus in this book to structural design smells (see [Figure 2.2](#)).



	Structural	Behavioral
Architecture		
Design	✓	
Implementation		

FIGURE 2.2

Scope of smells covered in this book.

The discussion on smells in this book is focused on popular object-orientation languages such as Java, C#, and C++. We also limit ourselves to language features supported in all three languages. For example, we do not cover multiple-inheritance in detail and limit as much as possible to single inheritance since multiple class inheritance is not supported in Java and C#.

Note that none of the design smells covered in this book are invented by us. All the design smells covered in this book have been presented or discussed earlier in research papers, books, or documentation of design analysis tools.

2.5 A CLASSIFICATION OF DESIGN SMELLS

When we set out to study structural design smells, we found 283 references to structural design smells in the existing literature. We realized that it is difficult to make sense of these smells unless we have a proper classification framework in place. In general, classification of entities improves human cognition by introducing hierarchical abstraction levels. Classification not only makes us mentally visualize the

entities, but also helps us differentiate them and understand them. Therefore, we wanted to create a classification framework for structural design smells.

2.5.1 DESIGN PRINCIPLES BASED CLASSIFICATION OF SMELLS

While we were exploring possible classification schemes for design smells, we realized that it would be very useful to developers if they were aware of the design principle that was violated in the context of a smell. Thus, we sought to classify design smells as a violation of one or more design principles. Another factor that influenced our use of this classification scheme is that if we can easily trace the cause of smells to violated design principles, we can get a better idea of how to address them.

For design principles, we use the *object model* (which is a conceptual framework of object orientation) of Booch et al. [47]. The four “major elements” of his object model are abstraction, encapsulation, modularization,³ and hierarchy. These are shown in Table 2.3. These principles are described in further detail in the introduction to Chapters 3, 4, 5, and 6.

We treat the four major elements of Booch’s object model as “design principles” and refer to them collectively as PHAME (Principles of Hierarchy, Abstraction, Modularization, and Encapsulation). These principles form the foundation of the smell classification scheme used in this book. Thus, each design smell is mapped to that particular design principle that the smell most negatively affects.

Table 2.3 High-level Principles Used in Our Classification

Design Principle	Description
Abstraction	The principle of abstraction advocates the simplification of entities through reduction and generalization: reduction is by elimination of unnecessary details and generalization is by identification and specification of common and important characteristics [48].
Encapsulation	The principle of encapsulation advocates separation of concerns and information hiding [41] through techniques such as hiding implementation details of abstractions and hiding variations.
Modularization	The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition.
Hierarchy	The principle of hierarchy advocates the creation of a hierarchical organization of abstractions using techniques such as classification, generalization, substitutability, and ordering.

2.5.2 NAMING SCHEME FOR SMELLS

We realized that in order to be useful to practitioners, our naming scheme for smells should be carefully designed. Our objectives here were four-fold:

³For the ease of naming smells, we have used the term *modularization* instead of the original term *modularity* used by Booch et al. [47].

- The naming scheme should be uniform so that a standard way is used to name all the smells
- The naming scheme should be concise so that it is easy for practitioners to recall
- Smells should be named such that it helps the practitioner understand what design principle was mainly violated due to which the smell occurred
- The name of a smell should qualify the violated design principle so that it gives an indication of how the smell primarily manifests

To achieve these objectives, we developed a novel naming scheme for design smells. In this naming scheme, each smell name consists of two words: an adjective (the first word) that qualifies the name of the violated design principle (the second word). Since the name of a smell consists of only two words, it is easy to remember. Further, specifying the violated design principle in the smell name allows a designer to trace back the cause of a smell to that design principle. This helps guide him towards adopting a suitable solution to address that specific smell. [Figure 2.3](#)

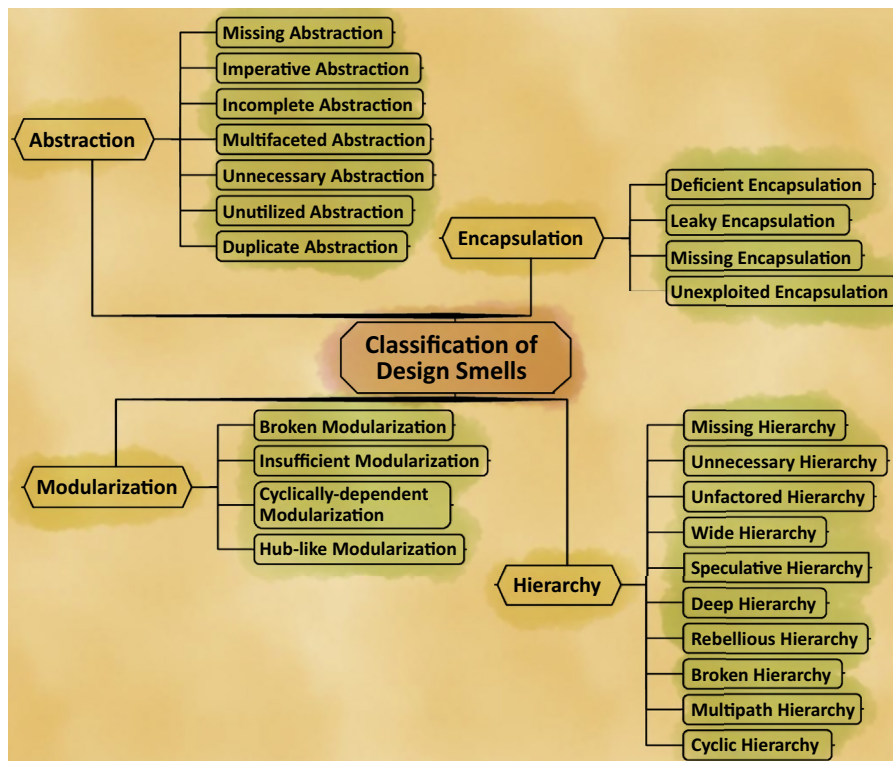


FIGURE 2.3

Classification of design smells.

illustrates our classification scheme (based on PHAME) and the naming scheme for smells covered in this book.

Note that these four design principles are not mutually exclusive. Therefore, the cause of some of the smells can be traced back to the violation of more than one design principle. In such situations, we leverage the enabling techniques for each principle, i.e., depending on the enabling technique it violates, we classify the smell under the corresponding principle. Let us discuss this using an example.

Consider the smell where data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions. We could argue that this smell arises due to wrong assignment of responsibilities across abstractions, therefore this smell should be classified under abstraction. We could also argue that since there is tight coupling between abstractions across which the data and/or methods are spread, this smell should be classified under modularization. To resolve this situation, we look at the enabling technique that this smell violates. In this case, this smell directly violates the enabling technique for modularization, “localize related data and methods,” hence we classify this smell under modularization.

2.5.3 TEMPLATE FOR DOCUMENTING SMELLS

For a consistent way of describing different design smells, we have adopted a uniform template to describe them. All the smells in this book have been documented using the template provided in [Table 2.4](#).

Table 2.4 Design Smell Template Used in This Book

Template Element	Description
Name & description	A concise, intuitive name based on our naming scheme (comprises two words: first word is an adjective, and second word is the primarily violated design principle). The name is followed by a concise description of the design smell (along with its possible forms).
Rationale	Reason/justification for the design smell in the context of well-known design principles and enabling techniques. (See Appendix A for a detailed list of design principles).
Potential causes	List of typical reasons for the occurrence of the smell (a nonexhaustive list based on our experience).
Example(s)	One or more examples highlighting the smell. If a smell has multiple forms, each form may be illustrated using a specific example.
Suggested refactoring	This includes generic high-level suggestions and steps to refactor the design smell, and a possible refactoring suggestion for each example discussed in the Examples section.

Table 2.4 Design Smell Template Used in This Book—cont'd

Template Element	Description
Impacted quality attributes	The design quality attributes that are <i>negatively</i> impacted because of this smell. The set of design quality attributes that are included for this discussion in the context of smells includes understandability, changeability, extensibility, reusability, testability, and reliability (see Table 2.1).
Aliases	Alternative names documented in literature that are used to describe the design smell. This includes variants, i.e., design smells documented in literature that are fundamentally identical to, but exhibit a slight variation from, the smell. The variation may include a special form, or a more general form of the design smell.
Practical considerations	Sometimes, in a real-world context, a particular design decision that introduces a smell may be purposely made either due to constraints (such as language or platform limitations) or to address a larger problem in the overall design. This section provides a non-exhaustive list of such considerations.

Chapters 3 to 6, respectively, cover smells that violate the principles of abstraction, encapsulation, modularization, and hierarchy.