CHAPTER

The Smell Ecosystem 7

Our study of design smells has revealed some interesting characteristics about them. If we were to consider the design as an ecosystem in which design decisions and smells coexist, we can see that smells both influence and are influenced by their ecosystem. Let us explore this further.

How does a smell manifest in design? A smell occurs as a result of a combination of one or more design decisions. In other words, the design ecosystem itself is responsible for the creation of the smell. The presence of the smell in turn impacts the ecosystem in several ways. First, it is likely that the presence of the smell triggers new design decisions that are needed to address the smell! Second, the smell can potentially influence or constrain future design decisions as a result of which one or more new smells may manifest in the ecosystem. Third, smells also tend to have an effect on other smells. For instance, some smells amplify the effects of other smells, or co-occur with or act as precursors to other smells. Clearly, smells share a rich relationship with the ecosystem in which they occur.

In this chapter, we discuss this *smell ecosystem*. We reflect upon the role of context in the identification and treatment of smells, and the interplay among smells.

7.1 THE ROLE OF CONTEXT

A software design can be described as a collection of design decisions. These design decisions include decisions about what classes should be included, how classes should behave, and how they should interact with each other. Each and every design decision is influenced by previously made design decisions, constraints on the design, and the requirements. In turn, every design decision also impacts the design; it can narrow down the set of future design decisions considerably or widen the scope of possible design decisions. In other words, each and every design decision impacts and even changes the context of the design.

Let us look at this from another perspective by considering an analogy from the medical domain. Most people who have taken medication for an illness would be familiar with the term "side effects." For instance, use of commonly available sleeping pills is usually accompanied with side effects such as headache, stomach pain, and constipation. Similarly, every design decision is accompanied by certain benefits and liabilities that it brings to the overall design. Sometimes, the benefits resulting from a design decision outweigh its liabilities, while at other times the liabilities

Refactoring for Software Design Smells. http://dx.doi.org/10.1016/B978-0-12-801397-7.00007-2

outweigh the benefits. To understand this better, let us ask what would happen if we apply an identical design decision to two different pieces of design. Would the impact on both designs be identical? Most likely, no! The impact would be different. For instance, a design decision to include a new class in one design may help improve the quality of that design, but the same decision might adversely affect the quality of a different design to which it is applied.

Clearly, design decisions should be made with full awareness of the benefits and liabilities on the overall design to ensure the high quality of the overall design. However, in practice, software developers tend to have an eye only on the benefits and ignore the liabilities of their design decisions. This introduces smells in their design.

Does this mean that if a design decision imposes a liability, it should not be made? No. On the other hand, awareness of the liabilities of a design decision brings to the fore an analysis and evaluation of the ways in which those liabilities can be addressed or diluted by introducing new design decisions. This process ensures that as the design evolves, its quality remains high.

What does all this mean? The take-away from the above discussion is that context plays a key role in software design. Similar to how the context has a significant influence on what design decisions are adopted, the context also determines several considerations with respect to design smells. We can explore this further by asking the questions below.

Q1. Are there cases in which a designer could use a smell acceptably because it achieves a larger purpose in the overall design? The answer is a definite YES. In fact, we have tried and captured many such cases in our catalog in the subsection "Practical Considerations" for each smell.

Q2. Are there cases when the violated principle behind a smell actually depends upon the context? The answer again is a definite YES. For instance, consider the case of two Graphical User Interface (GUI) editors shown in Figures 7.1 and 7.2. While these editors are independent of each other, they have a lot of similarities in terms of layout and features. For the sake of the discussion here, assume that the underlying code that supports these editors is similar. In such a case, if a feature change is needed across all editors, it will have to be made across the corresponding business logic of multiple editors. Clearly, it would be quite difficult to maintain this code.

What is the smell that we are discussing here? Depending upon the "context," this can be viewed as one of the following smells:

- *Duplicate Abstraction smell*: It can be argued that the classes underlying the two editors (let us call them `DefinitionEditor` and `ObservationEditor`, respectively) suffer from the "Identical Implementation" form of Duplicate Abstraction. A refactoring suggestion here would be to take the common code and put it into a new class named `CommonEditor`, and have the `DefinitionEditor` and `ObservationEditor` reference it for the common features.
- *Unfactored Hierarchy smell*: If `DefinitionEditor` and `ObservationEditor` are already part of a hierarchy and derive from a common base class (let us call it `Editor`) that does not contain the common features across the two editors, then

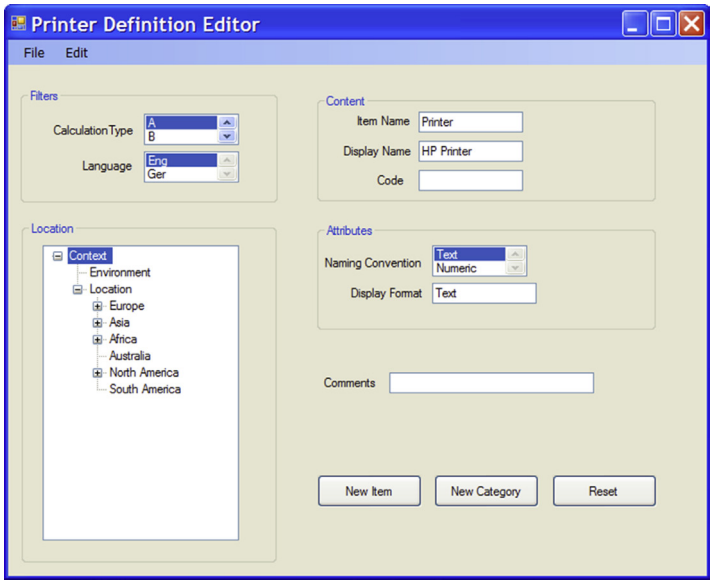

FIGURE 7.1

Printer Definition Editor user interface.

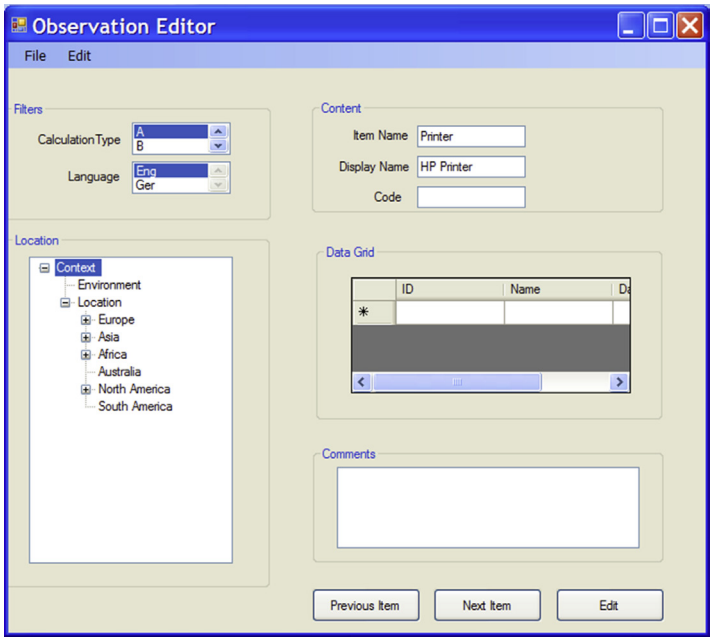

FIGURE 7.2

Observation Editor user interface.

it can be argued that an Unfactored Hierarchy smell is present in the design. A refactoring solution to address this smell would be to factor out the features common to these editors into the `CommonEditor` base class and encapsulate all the editor-specific unique features within the respective derived classes (i.e., `DefinitionEditor` and `ObservationEditor`) of `Editor`.

- *Unnecessary Abstraction smell*: Depending on the context, it can also be argued that there really is no need to create two separate classes (i.e., `DefinitionEditor` and `ObservationEditor`) corresponding to the two editors. Perhaps the design can contain only a single editor class that can be instantiated and configured with the right set of properties according to the specific requirements needed for an editor. In such a case, the `DefinitionEditor` and `ObservationEditor` classes become Unnecessary Abstractions and the refactoring solution to address this smell would be to remove these classes and instead have a single editor class.

As you can see, context plays an important role in determining if a design fragment exhibits a smell and, if so, what particular smell. The refactoring solution, similarly, not only depends on the identified smell but also on the context. It is quite possible that a refactoring unnecessarily complicates the design or even introduces another design smell if the refactoring does not consider the context. For instance, to refactor a smell, a designer may adopt a design pattern. However, if the forces behind the pattern do not match the forces that emerge from the context, the pattern may turn out to be a misfit for that context, and the liabilities of the pattern may outweigh its benefits. The use of this pattern may thus lead to the manifestation of new smells in the design.

Therefore, developers and designers need to carefully analyze the context and the overall design while determining smells and choosing the most appropriate refactoring to address them.

7.2 INTERPLAY OF SMELLS

Our experience with real-world projects has revealed that there is a strong interplay among design smells. For instance, smells often impact and interact with other smells in the design. They also tend to indicate deeper problems, such as architectural smells, in the design. In this section, we take a brief look at these interesting characteristics of smells.

7.2.1 SMELLS DO NOT OCCUR IN ISOLATION

Typically, multiple smells manifest together in a piece of design. We provide some examples below.

- Every experienced designer would have seen inheritance hierarchies that are "complex." Such complex hierarchies are often wide and deep making it quite

difficult to understand. Further, they also also have redundant inheritance paths and cycles making it difficult to comprehend the dependencies between types within the hierarchy. Such complex hierarchies often exhibit the following smells: Deep Hierarchy, Wide Hierarchy, Multipath Hierarchy, and Cyclic Hierarchy.

- Consider the dependency graph for some of the types in `java.util` library in Figure 7.3. The figure shows three smells: Cyclic Hierarchy (Section 6.10), Cyclically-dependent Modularization (Section 5.3), and Hub-like Modularization (Section 5.4). Classes `Calendar` and `Date` have large number of incoming and outgoing dependencies and hence suffer from Hub-like Modularization smell. Further, there is a Cyclically-dependent Modularization smell between the `TimeZone` and `Date` classes. Finally, there is a Cyclic Hierarchy smell between `Calendar` and `GregorianCalendar` class.
- Consider a design with a large and complex central class that has multiple responsibilities and is strongly coupled to satellite classes that have only data members. This design is quite common in real-world industrial projects and has been described as "abusive centralization of control" by Trifu and Marinescu [29]. Such a design exhibits various smells including Multifaceted Abstraction (Section 3.4), Broken Modularization (Section 5.1), and Insufficient Modularization (Section 5.2).

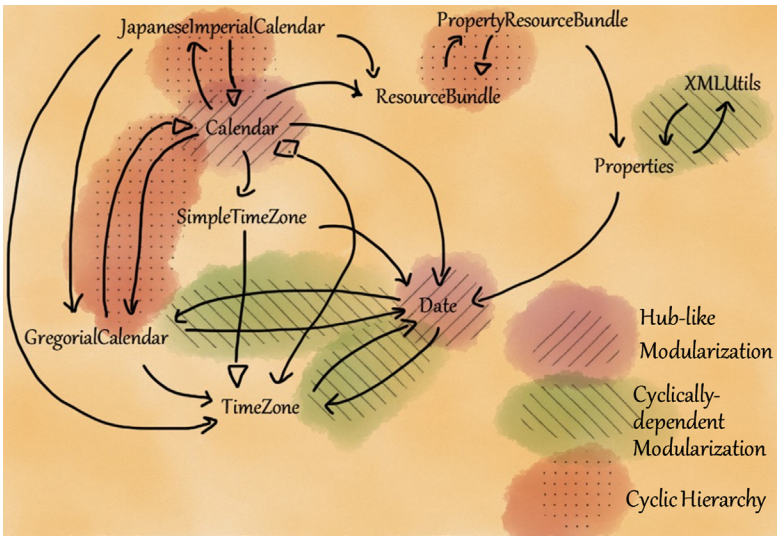

FIGURE 7.3

Smells found among some of the types in java.util library (not all dependencies shown).

While analyzing several such cases in real-world projects where multiple smells manifest together, we observed some interesting patterns emerging with respect to the interaction among smells. For instance, sometimes the effect of some smells is more pronounced in the presence of other smells, or sometimes smells co-occur or even act as precursors to other smells. We provide a brief overview of these types of interactions between smells in the following subsections.

7.2.1.1 Smells can amplify other smells

Design smells can amplify the effect of other smells. For instance, if there is a hierarchy suffering from a Deep Hierarchy smell, the presence of a supertype near the root that suffers from Insufficient Modularization can amplify the negative effects of the Deep Hierarchy smell. We will discuss how this occurs.

A subtype inherits methods from all its supertypes in a hierarchy and may override some of them. In a hierarchy suffering from Deep Hierarchy smell, overriding a method in the subtype toward the leaves poses the question: How do you know the semantics of the method that is to be overridden? One way is to look up the documentation comments for the method, but you will only get some information about its semantics and how to override it. Hence, you need to understand the method definitions all the way up to the type in which it originates. It is difficult to follow such a chain of overriding and the method definitions.

This problem is accentuated when a supertype near the root suffers from Insufficient Modularization. Consider the `javax.swing.ColorChooserDialog` hierarchy shown in Figure 7.4, which suffers from Deep Hierarchy. It should be noted that the `java.awt.Component` class is near the root of this hierarchy. Recall that this `Component` class suffers from Insufficient Modularization (Section 5.2) due to both bloated interface and bloated implementation (see Example 2 in Section 5.2). It has 332 methods, of which 259 are public, and the sum of the Cyclomatic complexity of its methods is 1143. One can only imagine the difficulty in comprehending the semantics of these methods, how they are interrelated, and how they are overridden in the multiple levels in the hierarchy if one were to try overriding them in the `ColorChooserDialog`!

Clearly, the presence of this `Component` class (which suffers from Insufficient Modularization) *amplifies* the negative effect of Deep Hierarchy. If the `Component` class were to be properly decomposed, then the effect of Deep Hierarchy smell in the `ColorChooserDialog` hierarchy would reduce.

7.2.1.2 Smells can co-occur

Some smells tend to co-occur with other smells. An example of commonly co-occurring smells is that of Insufficient Modularization with Multifaceted Abstraction. Recall that a class has Multifaceted Abstraction smell when it includes more than one responsibility. When the class is loaded with multiple responsibilities, its interface size, implementation complexity (or both) is likely to be high as well (indicating it has Insufficient Modularization). Hence, typically a class with Multifaceted Abstraction is likely to suffer from Insufficient Modularization as well.

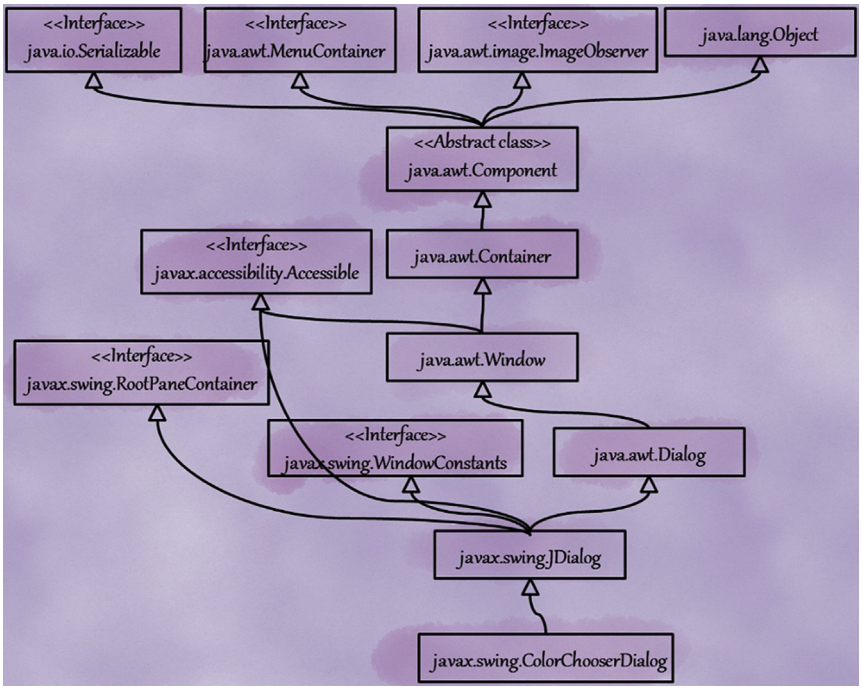

FIGURE 7.4

The ColorChooserDialog hierarchy.

The opposite is also true! A class with Insufficient Modularization is also likely to suffer from Multifaceted Abstraction. Due to this strong correlation between Insufficient Modularization and Multifaceted Abstraction, we term them *co-occurring* smells.

In fact, in practice, it is much easier to detect a class with Insufficient Modularization (using a static analysis tool) than to find one with Multifaceted Abstraction. The fact that they often co-occur can be leveraged to reduce the effort required in identifying each separately. This in turn reduces the time and effort required to improve the quality of the design.

It should be pointed out that once you commence the task of finding smells, you are likely to find quite a few smells hiding in your design. However, just because all these smells happen to be present in your design, it does not mean that they are co-occurring because they may not have a correlation.

7.2.1.3 Smells act as precursors to other smells

Depending on the design context, smells can often act as precursors to other smells. For instance, we have come across many cases in our experience where the presence of a Broken Modularization smell in a design has caused a Deficient Encapsulation smell! Recall that Broken Modularization arises when members that ideally

belong to a single abstraction are instead split and spread across multiple abstractions. Often in such a case, methods from these other abstractions need access to the data members of the original abstraction. To enable this, these data members that ideally should have private access are made publicly accessible, thus causing a Deficient Encapsulation smell. Similarly, it is not hard to imagine that a Broken Modularization smell can also lead to the occurrence of a Leaky Encapsulation smell.

We have seen that this relationship between Broken Modularization and Deficient Encapsulation occurs in contexts where the Broken Modularization involves the separation of the data members and methods, and not the separation of methods. In other words, the context of the design plays a key role in determining such causal relationships between smells.

Since the design context can vary widely, a precise identification of the different kinds of such causal relationships between smells would require extensive treatment, and is the subject of a different book altogether. We, however, encourage our readers to explore the smells in their designs and to learn from the different smell interactions that emerge from their exploration.

It is important also to add a note about addressing smells that have causal relationships. In cases where such causal relationships between smells manifest, when the root smell is addressed via proper refactoring, the smells that are caused due to the root smell also disappear along with it. However, the root smell is often not directly visible; in such a case, one has to start with a visible smell and slowly unravel the underlying smells one by one until the root smell is discovered.

7.2.2 SMELLS CAN INDICATE DEEPER PROBLEMS

An interesting aspect of smells is that often when you examine them, you may find a deeper underlying problem in the design. When we were looking for Cyclically-dependent Modularization smell in `java.awt` package, we found several cases of cyclic dependencies between classes. When we analyzed these cycles, we found that many of the cycles spanned across subpackages, indicating an architectural-level smell (see Figure 7.5). Addressing this architectural smell would likely require performing a large-scale refactoring to remove the cyclic dependencies between packages.

Other examples of possible deeper problems underlying the smells described in this book include:

- If you find Broken Hierarchy smell often in your designs, it could be indicative of the larger problem of using inheritance extensively for implementation reuse.
- If you find numerous classes with Imperative Abstraction smell, it may indicate the problem of "functional decomposition" (using procedural design in an object-oriented language).

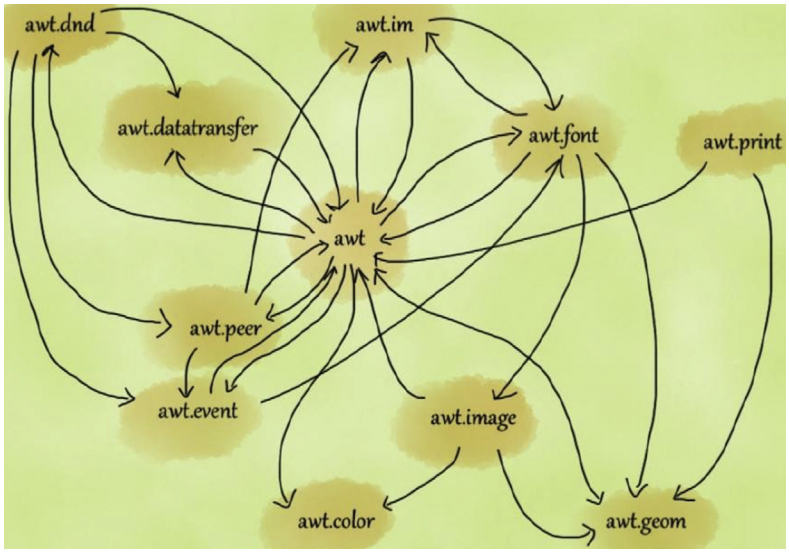

FIGURE 7.5

Cyclically-dependent Modularization smell at package-level in java.awt.

ANECDOTE

One of the authors was involved in the development of a software system that was structured using a layered architectural style. There were three layers: presentation layer, business layer, and data layer. The prescribed architecture clearly enunciated that the layering rules should not be violated; i.e., the layer on top should depend on the layers below but not vice versa. For instance, the presentation layer could use the services of the business and data layer; however, the data layer should not invoke the services of the business layer or presentation layer.

While analyzing the software, he found a cyclic dependency between the classes in the data layer and the presentation layer. When he explored further, he found that there were numerous references from the data layer to the presentation layer! Clearly, this meant that the actual architecture was violating the layering rules. This violation of layering is a well-known architectural smell. In this case, investigating the Cyclically-dependent Modularization smell led to the detection of an underlying architectural smell.

Resolving such deeper problems may yield a great benefit to the overall design. Hence, we recommend strongly that you not stop with just finding smells in your design; instead, you should use the discovered smells to unravel deeper problems in your design.