CHAPTER

3 Abstraction Smells

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.

Let us start with a trivial question: What is this figure? (Figure 3.1)

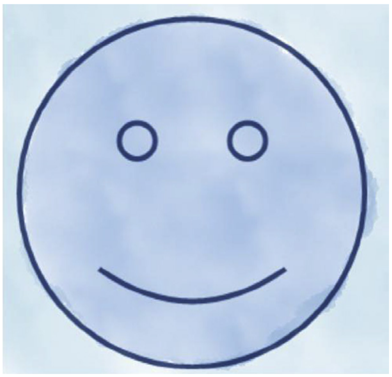

FIGURE 3.1

Human smiley face.

Of course it is a human smiley face, but how do we decide on that answer? We arrive at it through abstraction! There are hundreds of millions of human faces and each and every face is unique (there are some exceptions). How are we able to cope with this complexity? We eliminated non-essential details such as hair styles and color. We also arrived at the answer by generalizing commonalities such as every face has two eyes and when we smile our lips curve upwards at the ends.

Abstraction is a powerful principle that provides a means for effective yet simple communication and problem solving. Company logos and traffic signs are examples of abstractions for communication. Mathematical symbols and programming languages are examples of abstraction as a tool for problem solving.

Modern software is so complex that it often spans millions of lines of code. To tame the complexity of such systems, we need to apply powerful principles such as abstraction in the most effective way possible. How do we apply the principle of abstraction in software design?

We list below some key enabling techniques that we have gleaned from our experience that allow us to apply the principle of abstraction in software design (Figure 3.2):

- **Provide a crisp conceptual boundary and an identity**. Each abstraction should have a crisp and clear conceptual boundary and an identity. For instance, instead of "passing around" a group of data values representing a date, coordinates of a rectangle, or attributes of an image, they can be created as separate abstractions in the code.
- **Map domain entities**. There should be a mapping of vocabulary from the problem domain to the solution domain, i.e., objects recognized by the problem domain should be also represented in the solution domain. For instance, if there is a clip art that you can insert in a word processor application, it is easier to comprehend the design when it also has a corresponding abstraction named ClipArt.
- **Ensure coherence and completeness**. An abstraction should completely support a responsibility. For instance, consider a class representing a combo box. If the class supports disabling or hiding certain elements, it should provide an option for enabling or showing those elements as well; providing some of the methods and leaving a few would leave an abstraction incoherent and incomplete.
- **Assign single and meaningful responsibility**. Ensure that each abstraction has a unique non-trivial responsibility assigned to it. For instance, a class representing an

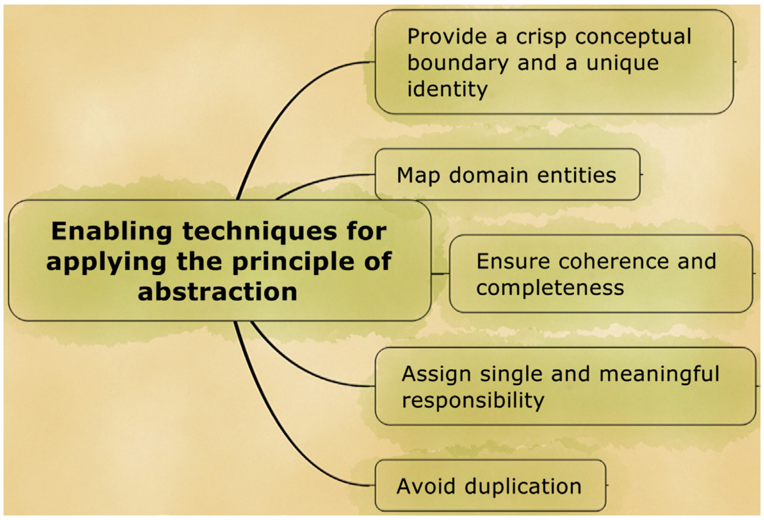
Enabling techniques for applying the principle of abstraction
Provide a crisp conceptual boundary and a unique identity
Map domain entities
Ensure coherence and completeness
Assign single and meaningful responsibility
Avoid duplication

FIGURE 3.2

Enabling techniques for applying abstraction.

image should not be overloaded with methods for converting one kind of image representation format to another format. Furthermore, it does not make sense to create a class named `PI` that serves as a placeholder for holding the value of the constant π.

- **Avoid duplication**. Ensure that each abstraction—the name as well as its implementation—appears only once in design. For instance, you may require creating a class named `List` when designing a list control in Graphical User Interfaces, a linked-list data structure, or for maintaining a TO-DO list; however, naming all these abstractions as `List` will confuse the users of your design. Similarly, implementation duplication (copying the code for priority queue data structure for implementing a scheduler) will introduce additional effort in maintaining two pieces of the same code.

Interestingly, each smell described in this chapter maps to a violation of an enabling technique. Figure 3.3 gives an overview of the smells that violate the

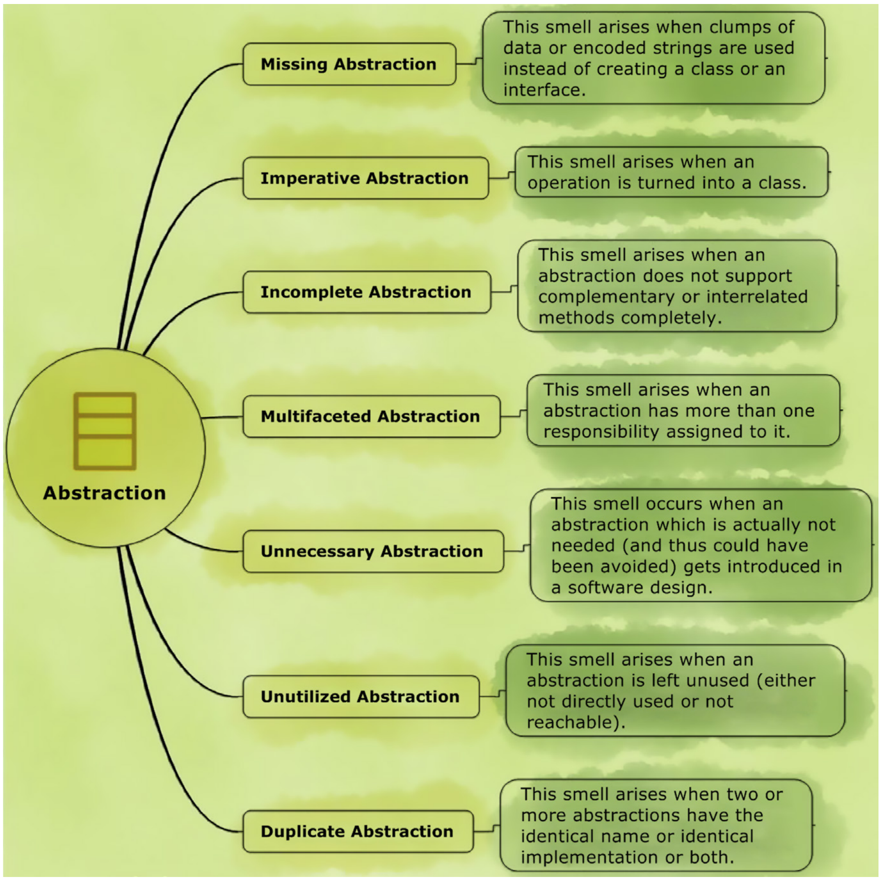
Abstraction
Missing Abstraction
This smell arises when clumps of data or encoded strings are used instead of creating a class or an interface.
Imperative Abstraction
This smell arises when an operation is turned into a class.
Incomplete Abstraction
This smell arises when an abstraction does not support complementary or interrelated methods completely.
Multifaceted Abstraction
This smell arises when an abstraction has more than one responsibility assigned to it.
Unnecessary Abstraction
This smell occurs when an abstraction which is actually not needed (and thus could have been avoided) gets introduced in a software design.
Unutilized Abstraction
This smell arises when an abstraction is left unused (either not directly used or not reachable).
Duplicate Abstraction
This smell arises when two or more abstractions have the identical name or identical implementation or both.

FIGURE 3.3

Smells resulting due to the violation of the principle of abstraction.

Table 3.1 Design Smells and the Abstraction Enabling Techniques They Violate

Design Smell (s)	Violated Enabling Technique
Missing Abstraction (see Section 3.1)	Provide a crisp conceptual boundary and a unique identity
Imperative Abstraction (see Section 3.2)	Map domain entities
Incomplete Abstraction (see Section 3.3)	Ensure coherence and completeness
Multifaceted Abstraction, Unnecessary Abstraction, Unutilized Abstraction (see Sections 3.4, 3.5, 3.6)	Assign single and meaningful responsibility
Duplicate Abstraction (see Section 3.7)	Avoid duplication

principle of abstraction, and Table 3.1 provides an overview of mapping between these smells and the enabling technique(s) they violate. A detailed explanation of how these smells violate enabling techniques is discussed in the Rationale subsection of each smell description. Note that when we use the term "abstraction" to refer to a design entity, it means a class or an interface unless otherwise explicitly stated.

In the rest of this chapter, we'll discuss the specific smells that result due to the violation of the principle of abstraction.

3.1 MISSING ABSTRACTION

This smell arises when clumps of data or encoded strings are used instead of creating a class or an interface.

3.1.1 RATIONALE

An enabling technique for applying the principle of abstraction is to *create entities with crisp conceptual boundaries and a unique identity*. Since the abstraction is not explicitly identified and rather represented as raw data using primitive types or encoded strings, the principle of abstraction is clearly violated; hence, we name this smell Missing Abstraction.

Usually, it is observed that due to the lack of an abstraction, the associated data and behavior is spread across other abstractions. This results in two problems:

- It can expose implementation details to different abstractions, violating the principle of encapsulation.
- When data and associated behavior are spread across abstractions, it can lead to tight coupling between entities, resulting in brittle and non-reusable code. Hence, not creating necessary abstractions also violates the principle of modularization.

3.1.2 POTENTIAL CAUSES

Inadequate design analysis

When careful thought is not applied during design, it is easy to overlook creating abstractions and use primitive type values or strings to "get the work done." In our experience, this often occurs when software is developed under tight deadlines or resource constraints.

Lack of refactoring

As requirements change, software evolves and entities that were earlier represented using strings or primitive types may need to be refactored into classes or interfaces. When the existing clumps of data or encoded strings are retained as they are without refactoring them, it can lead to a Missing Abstraction smell.

Misguided focus on minor performance gains

This smell often results when designers compromise design quality for minor performance gains. For instance, we have observed developers using arrays directly in the code instead of creating appropriate abstractions since they feel that indexing arrays is faster than accessing members in objects. In most contexts, the performance gains due to such "optimizations" are minimal, and do not justify the resultant trade-off in design quality.

3.1.3 EXAMPLES

Example 1

Consider a library information management application. Storing and processing ISBNs (International Standard Book Numbers) is very important in such an application. It is possible to encode/store an ISBN as a primitive type value (long integer/decimal type) or as a string. However, it is a poor choice in this application. Why? To understand that, let's take a quick look at ISBNs.

ISBN can be represented in two forms—10-digit form and 13-digit form—and it is possible to convert between these two forms. The digits in an ISBN have meaning; for example, an ISBN-13 number consists of these elements: *Prefix Element*, *Registration Group Element*, *Registrant Element*, *Publication Element*, and *Checksum*. The last digit of an ISBN number is a checksum digit, which is calculated as follows: starting from the first digit, the values of the odd digits are kept the same, and the values of even numbered digits are multiplied by three; the sum of all the values modulo 10 is the value of the last digit. So, given a 10 or 13 digit number, you can validate whether the given number is a valid ISBN number. ISBN numbers can be converted to barcodes, and a barcode processor can recognize an ISBN number.

Implementation of a library information management application involves logic that accepts, validates, processes, or converts between ISBN numbers. It is possible to encode ISBN numbers as strings or as a primitive type value in such an application. However, in such a case, the logic that processes the numbers will be spread as well as duplicated in many places. In the context of a library information system that has considerable logic involving ISBN numbers, not encapsulating ISBN numbers as class(es) indicates a Missing Abstraction smell.

Example 2

Applications are often characterized by clumps of primitive type data values that are always used together. In many cases, these data clumps indicate a Missing Abstraction. Consider a drawing application that allows a user to select and manipulate a rectangular region of an image. One (naïve) way to represent this rectangular region is to either use two pairs of values—say variables of double type (x1, y1) and (x2, y2)—or variables of double type (x1, y1) and (height, width). These values will always be used together and passed along to multiple methods. Such "data clumps" [7] indicate a Missing Abstraction.

Example 3

Strings are often used to encode information. In the case of APIs (Application Programming Interface), the problem with encoding data in strings is that once the API is released, it is very difficult to change the encoding format since clients that depend on the API will be affected. Let us take a look at a detailed example from Java Development Kit (JDK).

From 1.0 version of Java, the stack trace was printed to the standard error stream as a string with `printStackTrace()` method:

```
public class Throwable {
  public void printStackTrace();
  // other methods elided.
}
```

Client programs that needed programmatic access to the stack trace elements had to write code to process the stack trace to get, for example, the line numbers or find if the bug is a duplicate of another already-filed bug. Due to this dependence of client programs on the format of the string, the designers of JDK were forced to retain the string encoding format in future versions of JDK.

3.1.4 SUGGESTED REFACTORING

The refactoring for this smell is to create abstraction(s) that can internally make use of primitive type values or strings. For example, if a primitive type value is used as a "type-code," then apply "replace type-code with class"; furthermore, create or move operations related to the data such as constructors, validation methods, and copying methods within the new class. For a detailed discussion of refactoring for this smell, see Kerievsky's book [50].

Suggested refactoring for Example 1

The example discussed the need to handle different kinds of ISBN numbers such as ISBN-10 and ISBN-13. A potential refactoring would be to abstract ISBN as an abstract class or interface with common operations in it. `ISBN-10` and `ISBN-13` can be subclasses that extend the `ISBN` supertype (see Figure 3.4).

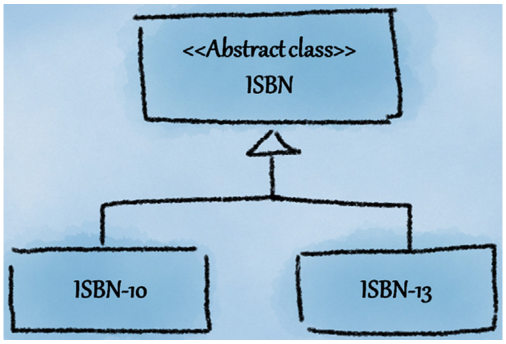

FIGURE 3.4

Suggested refactoring for ISBN example (Example 1).

Suggested refactoring for Example 2

Revisiting the example of the drawing application, a refactoring suggestion would be to abstract the required fields into a new class—say `Rectangle` class or `SelectedRegion` class—and move methods operating on these fields to the new class.

Suggested refactoring for Example 3

In the context of the stack trace example, a refactoring suggestion is that instead of first encoding stack trace elements as strings and then exposing them to clients, these stack trace elements should be exposed programmatically. The Java API was improved in version 1.4 to have programmatic access to the stack trace through the introduction of `StackTraceElement` class. Note that even though a new method is added, the method `printStackTrace()` and the format of the stack trace has been retained to support the existing clients.

```
public class Throwable {
  public void printStackTrace();
  public StackTraceElement[] getStackTrace(); // Since 1.4
  // other methods elided.
}
```

The `StackTraceElement` is the "Missing Abstraction" in the original design. It was introduced in Java 1.4 as follows:

```
public final class StackTraceElement {
  public String getFileName();
  public int getLineNumber();
  public String getClassName();
  public String getMethodName();
  public boolean isNativeMethod();
}
```

3.1.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—When a key entity is not represented as an abstraction and the logic that processes the entity is spread across the code base, it becomes difficult to understand the design.
- **Changeability** and **Extensibility**—It is difficult to make enhancements or changes to the code when relevant abstractions are missing in design. First, it is difficult even to figure out the parts of the code that need to be modified to implement a change or enhancement. Second, for implementing a single change or enhancement, modifications need to be made at multiple places spread over the code base. These factors impact the changeability and extensibility of the design.
- **Reusability** and **Testability**—Since some abstractions that correspond to domain or conceptual entities are missing, and the logic corresponding to the entities is spread in the code base, both reusability and testability of the design are impacted.
- **Reliability**—An abstraction helps provide the infrastructure to ensure the correctness and integrity of its data and behavior. In the absence of an abstraction, the data and behavior is spread across the code base; hence, integrity of the data can be easily compromised. This impacts reliability.

3.1.6 ALIASES

This smell is also known in literature as:

- "Primitive obsession" [7]—This smell occurs when primitive types are used for encoding dates, currency, etc. instead of creating classes.
- "Data clumps" [7]—This smell occurs when there are clumps of data items that occur together in lots of places instead of creating a class.

3.1.7 PRACTICAL CONSIDERATIONS

Avoiding over-engineering

Sometimes, entities are merely data elements and don't have any behavior associated with them. In such cases, it may be over-engineering to represent them as classes or interfaces. Hence, a designer must carefully examine the application context before deciding to create an explicit abstraction. For example, check if the following are needed (a non-exhaustive list) to determine if creating an abstraction is warranted:

- default initialization of data values using constructors
- validation of the data values

- support for pretty printing the data values
- acquired resources (if any) are to be released

3.2 IMPERATIVE ABSTRACTION

This smell arises when an operation is turned into a class. This smell manifests as a class that has only one method defined within the class. At times, the class name itself may be identical to the one method defined within it. For instance, if you see class with name `Read` that contains only one method named `read()` with no data members, then the `Read` class has Imperative Abstraction smell. Often, it is also seen in the case of this smell that the data on which the method operates is located within a different class.

It should be noted that it is sometimes desirable to turn an operation into a class. Section 3.2.7 covers this aspect in greater detail.

3.2.1 RATIONALE

The founding principle of object-orientation is to capture real world objects and represent them as abstractions. By following the enabling technique *map domain entities*, objects recognized in the problem domain need to be represented in the solution domain, too. Furthermore, each class representing an abstraction should encapsulate data and the associated methods. Defining functions or procedures explicitly as classes (when the data is located somewhere else) is a glorified form of structured programming rather than object-oriented programming. One-operation classes cannot be representative of an "abstraction," especially when the associated data is placed somewhere else. Clearly, this is a violation of the principle of abstraction. Since these classes are 'doing' things instead of 'being' things, this smell is named Imperative Abstraction.

If operations are turned into classes, the design will suffer from an explosion of one-method classes and increase the complexity of the design. Furthermore, many of these methods that act on the same data would be separated into different classes and thus reduce the cohesiveness of the design. For these reasons, this smell also violates the principles of encapsulation and modularization.

3.2.2 POTENTIAL CAUSES

Procedural thinking

A common cause for this smell is procedural thinking in an object-oriented setup. One of the authors was once involved in a project where a programmer from C background had designed a system in Java. In the resultant design, data that would have been encapsulated within "structs" in C was mapped to Java classes containing only public data members. The operations on the data were encapsulated in *separate* Java classes! The procedural thinking of his C experience resulted in the design of a procedural program that was, however, implemented in an object-oriented language.

3.2.3 EXAMPLES

Example 1

Consider the case of a large-sized financial application. This application employs classes named `CreateReport`, `CopyReport`, `DisplayReport`, etc. to deal with its report generation functionality. Each class has exactly one method definition named `create`, `copy`, `display`, etc., respectively, and suffers from Imperative Abstraction smell. The data items relating to a report such as name of the report, data elements that need to be displayed in the report, kind of report, etc. are housed in a "data class" named `Report`.

The smell not only increases the number of classes (in this case there are at least four classes when ideally one could have been used), but also increases the complexity involved in development and maintenance because of the unnecessary separation of cohesive methods (Figure 3.5).

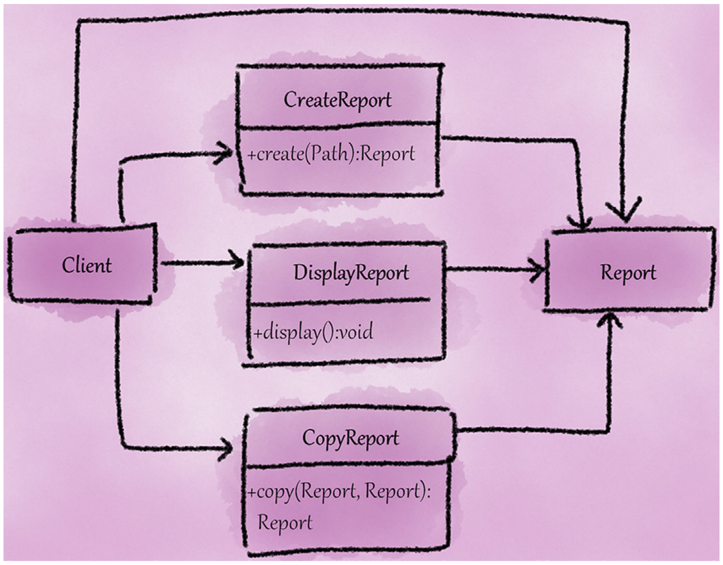

FIGURE 3.5

Class diagram for the report generation functionality (Example 1).

Example 2

Figure 3.6 shows a small fragment of the design of an image processing application. The objective behind the classes shown in the figure is to compare two images and store the differences in the images. The `Application` class uses the `calculateOffset()` method in `OffsetCalculator` class to calculate the offset between two given images. The calculated offset is returned to the `Application`, which then invokes the `saveOffset()` method in `SaveHandler` class to store it.

A deeper look at the `OffsetCalculator` and `SaveHandler` classes shows that they *only* contain one method each, namely `calculateOffset()` and `saveOffset()`, respectively. Thus, the `OffsetCalculator` and `SaveHandler` classes suffer from the Imperative Abstraction smell.

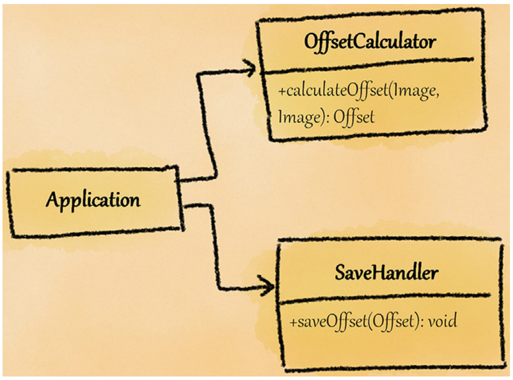
OffsetCalculator
+calculateOffset(Image, Image): Offset
Application
SaveHandler
+saveOffset(Offset): void

FIGURE 3.6

Class diagrams for the image processing application (Example 2).

3.2.4 SUGGESTED REFACTORING

To refactor the Imperative Abstraction design smell, you have to either find or create an appropriate abstraction to house the method existing within the Imperative Abstraction. You also have to encapsulate the data needed by the method within the same abstraction to improve cohesion and reduce coupling.

Suggested refactoring for Example 1

For the report generation part of the financial application, a suggested refactoring is to move the methods in each of the classes suffering from Imperative Abstraction to the `Report` class itself (see Figure 3.7). Moving all the report-related operations to the `Report` class makes the `Report` class a proper "abstraction" and also removes the Imperative Abstraction smell. The design becomes cleaner and less complex.

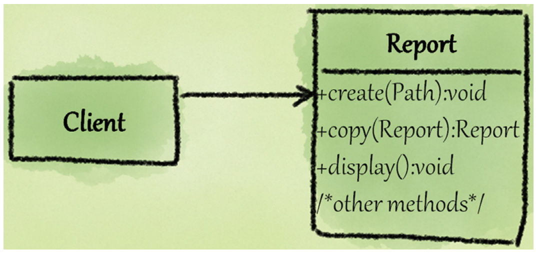

FIGURE 3.7

Suggested refactoring for the report generation functionality (Example 1).

Suggested refactoring for Example 2

For the image processing application, a potential refactoring solution is to use an `ImagePair` class that encapsulates the offset calculation and offset saving operations. This is shown in Figure 3.8.

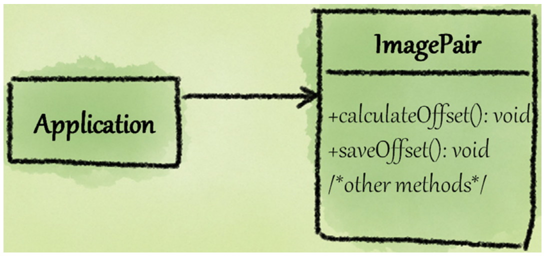

FIGURE 3.8

Suggested refactoring for the image processing application (Example 2).

A question that may arise here is, "Are we not mixing up the responsibilities of calculating the offset and saving it, within a single class? Are we not violating Single Responsibility Principle?" To address this, one way to think about this design is that the actual opening of a file or setting up a connection to a database to save the offset is going to be encapsulated in a different class (not shown here). So the `saveOffset()` method is going to invoke that other class. In other words, the responsibility of "saving the offset" is not really getting mixed in the `ImagePair` class.

3.2.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—An abstraction with this smell does not have a direct mapping to the problem domain; it only represents a part of the behavior of a domain entity but does not represent the entity itself. Thus understandability of the abstraction is poor. Having numerous single-method classes leads to "class explosion" and complicates the design, and thus impacts the understandability of the overall design.
- **Changeability** and **Extensibility**—The presence of an Imperative Abstraction smell does not impact the changeability and extensibility of the abstraction itself. However, if a modification or enhancement is required to a domain or conceptual entity (which is realized by multiple abstractions), it will require changes to multiple abstractions throughout the design. Thus, the changeability and extensibility of the overall design are impacted.
- **Reusability**—Consider the report generation functionality that was discussed in Example 1. If we want to reuse this functionality in a different context, we would have to take the `CreateReport`, `CopyReport`, and `DisplayReport`

classes along with the class that serves to hold the report data, and adapt them to the new context. Clearly, this would require much more effort than if we were to reuse the `Report` class (discussed in Suggested refactoring for Example 1).

- **Testability**—If an abstraction with Imperative Abstraction smell is self-sufficient, it is easy to test it. However, typically, in designs that have this smell, multiple abstractions with single operations need to be collectively tested. This impacts testability of the overall design.

ANECDOTE

One of the anecdotes we came across in the Smells Forum concerned a toolkit development project written in Java. Just before the release of the product, experts external to the team performed a systematic design review to evaluate the design quality of the software.

During the review, it was observed that the name of a class was `Diff` and it had only one method named `findDiff()`! The `findDiff()` method would take two versions of a csv (comma separated values) file and report the difference between the files. It was obvious that the class had Imperative Abstraction smell.

A deeper look at the design revealed that most classes had names starting with a verb such as "generate," "report," "process," and "parse." These classes mostly had one public method with other methods serving as private helper methods. There was almost no use of class inheritance (interface inheritance was used, but mainly for implementing standard interfaces such as `Comparable` and `Cloneable`). This was a bigger finding—the design followed "functional decomposition" and not "object-oriented decomposition"!

Reflecting on the toolkit design revealed that object-oriented decomposition could have been more beneficial for the project, since such an approach would have helped making adaptations to variations as well as implementing new features easier. Since the toolkit consisted of tools that "do" things, developers unconsciously designed and evolved the software using a "functional decomposition" approach.

The design review results surprised the development team! They had never anticipated that the cause for one of the key structural problems in their project was a "procedural programming" mindset. Even though the team consisted of experienced designers, a major problem like this was overlooked.

However, changing the toolkit to follow an object-oriented decomposition approach would require rearchitecting the whole toolkit. Since there were already a large number of customers using the toolkit, it was a risky endeavor to take-up this reengineering activity. Hence, the management decided to live with the existing design.

Reflecting on this anecdote highlights the following key insights:

- Even when a person is part of and is familiar with a project, it is easy for him to overlook obvious design problems. Our experience shows that systematic and periodic design reviews by experts (external to the project) help identify opportunities for improving the design quality of the software [4].
- The presence of certain smells often indicates a much deeper problem in design. In this case, unraveling the cause behind the Imperative Abstraction smell revealed "functional decomposition" to be the deeper problem.
- It is ironic that the most critical problems are often found just before a release. It is very risky to "fix" the design so late in such cases. So, learning about smells and avoiding them throughout the development cycle is important in maintaining the quality of the design and avoiding technical debt.

3.2.6 ALIASES

This smell is also known in literature as:

- "Operation class" [51,52]—This smell occurs when an operation that should have been a method within a class has been turned into a class itself.

3.2.7 PRACTICAL CONSIDERATIONS

Reification

"Reification" is the promotion or elevation of something that is not an object into an object. When we reify behavior, it is possible to store it, pass it, or transform it. Reification improves flexibility of the system at the cost of introducing some complexity [52].

Many design patterns [54] employ reification. Examples:

- State pattern: Encoding a state-machine.
- Command pattern: Encoding requests as command objects. A permitted exception for this smell is when a Command pattern has been used to objectify method requests.
- Strategy pattern: Parameterizing a procedure in terms of an operation it uses.

In other words, when we consciously design in such a way to elevate non-objects to objects for better reusability, flexibility, and extensibility (i.e., for improving design quality), it is not a smell.

3.3 INCOMPLETE ABSTRACTION

This smell arises when an abstraction does not support complementary or interrelated methods completely. For instance, the public interface of an abstraction may provide an `initialize()` method to allocate resources. However, a `dispose()` method that will allow clean-up of the resources before it is deleted or re-collected[1] may be missing in the abstraction. In such a case, the abstraction suffers from Incomplete Abstraction smell because complementary or interrelated methods are not provided completely in its public interface.

3.3.1 RATIONALE

One of the key enabling techniques for abstraction is to "*create coherent and complete abstractions*." One of the ways in which coherence and completeness of an abstraction may be affected is when interrelated methods are not supported by the abstraction. For example, if we need to be able to add or remove elements in a data structure, the type abstracting that data structure should support both `add()` and

[1] In languages that support garbage collection.

`remove()` methods. Supporting only one of them makes the abstraction incomplete and incoherent in the context of those interrelated methods.

If an abstraction supports such interrelated methods only partially, it may force the users of the abstraction to implement the rest of the functionality. Sometimes, to overcome the problem of such an incomplete interface exposed by an abstraction, clients attempt to access the internal implementation details of the abstraction directly; in this way, encapsulation may be violated as a side-effect of not applying the principle of abstraction properly. Further, since one of the interrelated methods is implemented somewhere else, cohesion is compromised and the *Single Responsibility Principle* (*SRP*) (which per definition requires a responsibility to be *completely* encapsulated by a class) is violated. Since complementary or interrelated methods are not provided completely in the abstraction, this smell is named Incomplete Abstraction.

3.3.2 POTENTIAL CAUSES

Missing overall perspective

When designers or developers create abstractions to add new features, their focus is usually limited to the actual specific requirements that need to be supported. In this process, they overlook if the abstraction supports all the interrelated methods completely or not.

Not adhering to language or library conventions

Consider the container `java.util.HashMap` in JDK. For retrieving an element based on the given input key, the `get()` method first compares the hash code of the input key (using the `hashCode()` method) with the mapped key found in the hash map; next, it invokes `equals()` method on the input key to check if it equals the mapped key. Now, if a class does not override `hashCode()` and `equals()` methods together and the objects of that class are used in `HashMap`, it is possible that the retrieval mechanism will not work correctly. Hence, one of the conventions required by the Java library is that these two methods should be overridden together in a class when the objects of that class are used with hash-based containers.

Such conventions also exist in other languages such as C# and C++. For instance, in C++, it is better to define either all of default constructor, copy constructor, assignment operator, and destructor (virtual destructor if the class would be used polymorphically) or none. If there are any missing definitions (e.g., virtual destructor), the abstraction is incomplete and could result in runtime problems.

Sometimes developers overlook such language or library conventions, and provide only partial set of methods. In such cases, it leads to Incomplete Abstraction smell.

3.3.3 EXAMPLES

Example 1

An interesting instance of “Incomplete Abstraction” is observed in JDK’s `javax.swing.ButtonModel` interface. It provides `setGroup()` method, which according to its documentation, "identifies the group the button belongs to—needed for radio buttons, which are mutually exclusive within their group." The `ButtonModel` interface does not provide the symmetric `getGroup()` method and hence suffers from Incomplete Abstraction smell.

Example 2

The *Abstract Factory* pattern [54] is a creational pattern that allows families of objects to be created. Figure 3.9 shows the solution structure of the pattern. Interestingly, the abstract factory as defined by the pattern does not specifically include the responsibility to delete the objects they have created. While this may not be needed in languages that support garbage collection, we have encountered several cases in real-world projects where an Abstract Factory pattern has been used correctly but object deletion has been overlooked by designers since it is not explicitly addressed in the pattern. Thus, if the factory classes shown in Figure 3.9 were to be used as they are, the resulting design would exhibit Incomplete Abstraction smell.

ANECDOTE

An attendee at one of the design forums shared an interesting anecdote that is relevant here. He was working on the development of a software tool that facilitated the configuration of smart devices. A week before the tool's release, it was observed that the tool was characterized by a number of memory leaks. To quickly identify and address the cause of these runtime memory leaks, an expert on memory leaks was called in as a consultant.

Based on past experience, the consultant could guess that the reason for memory leaks was that there were some objects that were not getting cleaned up properly. So, he started examining how and when objects were getting deleted at runtime. He found out that while there was a systematic procedure to create objects, there was little or no attention paid to how those created objects were eventually deleted. Most developers had seemingly completely ignored the deletion of created objects!

To understand the reason, the consultant decided to dig deeper. He found out that the source of the problem was that the project architect had only listed "creation" methods in the design document that he had handed to the developers. When queried, the project architect said that he expected the software developers to "understand" that a method marked for object creation in his design was actually a placeholder for both "creation and deletion" methods. He stated that it is obvious that objects that are created need to be cleaned up and, therefore, the software developers writing the code should have taken care of object deletion as well.

When the consultant asked the software developers responsible for code construction about this, they said that they are told to "strictly follow the design drafted by the architect" so that there is no architecture erosion or over-engineering on their part. Therefore, they tended to overlook or seldom care about concerns not explicitly captured in the design.

There are two insights that emerge from this anecdote:

- Architects should be extremely careful and diligent about what is communicated in the design document. A stringent review of design is needed to expose issues such as the ones mentioned above.
- A healthy environment needs to exist in the project team that allows developers to give feedback to the architect(s) about the prescribed design. In fact, new age development methodologies such as those based on Agile promote the idea of "collective ownership" of the design and architecture. This means making all the team members responsible for the design and involving all of them in the design decisions that are made so that everyone's understanding is enhanced and a high-quality design emerges.

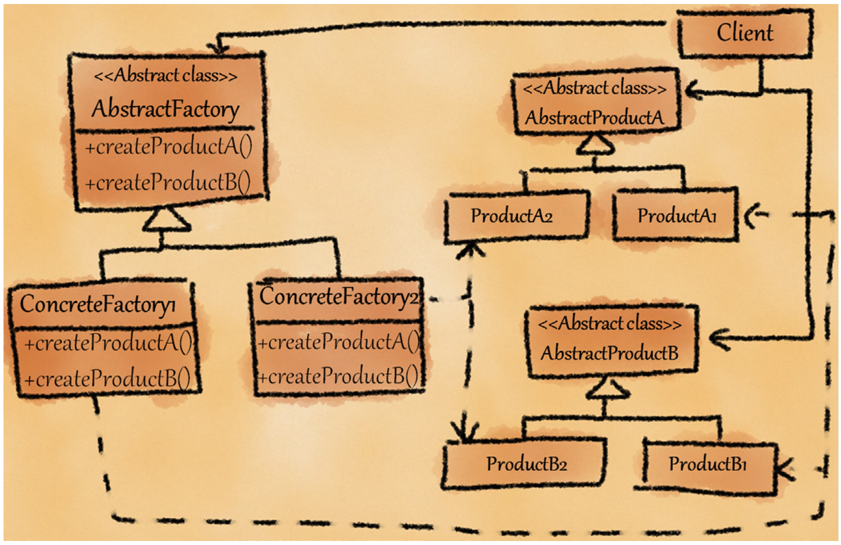

FIGURE 3.9

Class diagram for the Abstract Factory pattern [54].

3.3.4 SUGGESTED REFACTORING

If you find interfaces or classes that are missing "complementary and symmetric" methods, it possibly indicates an Incomplete Abstraction. Table 3.2 lists some of the common pairs of operations. Note that these merely serve as exemplars of common usage and the actual names will vary depending on the context. For example, the names of operations in the context of a `Stack` would be `push` and `pop`, whereas in the context of data streams, it would be `source` and `sink`. The refactoring strategy for this smell is to preferably introduce the missing complementary operation(s) in the interface or class itself.

Also, if there are any language or library conventions that require providing interrelated methods together, the refactoring is to add any such missing methods in the abstraction.

Table 3.2 Some Common Symmetric Method Pairs

Min/max	Open/close	Create/destroy	Get/set
Read/write	Print/scan	First/last	Begin/end
Start/stop	Lock/unlock	Show/hide	Up/down
Source/target	Insert/delete	First/last	Push/pull
Enable/disable	Acquire/release	Left/right	On/off

Suggested refactoring for Example 1

The refactoring for the `ButtonModel` example from JDK ideally involves defining the `getGroup()` method in the `ButtonModel` interface itself. However, since JDK is a public API, adding a method to an interface would break the existing classes that implement that interface (remember that all methods declared in an interface must be defined in a class that implements the interface). Hence, to avoid breaking existing clients, the `getGroup()` method was added in its derived class `DefaultButtonModel` in JDK version 1.3 (see Figure 3.10).

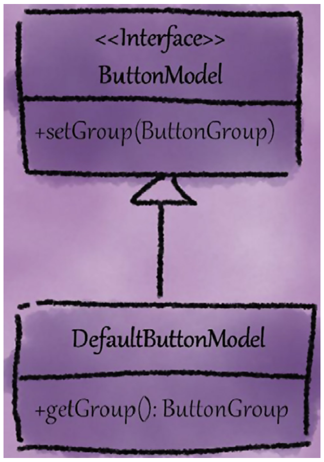

FIGURE 3.10

Refactoring for ButtonModel interface in JDK 1.3 (Example 1).

So, the lesson that we learn from this example is that it is difficult to evolve interfaces, and hence it is important to be aware of and avoid smells such as Incomplete Abstraction when designing APIs.

Suggested refactoring for Example 2

The suggested refactoring for the Abstract Factory example would be to ensure that, when the context requires, the responsibility of object deletion is also explicitly included within the `AbstractFactory` class [19]. This will allow creation and deletion of objects to be handled within a single class. This decreases the complexity arising from addressing object creation and deletion mechanisms in separate classes, and improves the maintainability of the system.

It should be noted, in the same way, a Factory Disposal Method pattern [19] can be used to help delete created objects in the context of Factory Method pattern [54].

3.3.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—Understandability of the abstraction is adversely impacted because it is difficult to comprehend why certain relevant method(s) are missing in the abstraction.
- **Changeability** and **Extensibility**—Changeability and extensibility of the abstraction are not impacted. However, when complementary methods are separated across abstractions, changes or enhancements to one method in an abstraction may impact the other related abstractions.
- **Reusability**—If some of the operations are not supported in an abstraction, it is harder to reuse the abstraction in a new context because the unsupported operations will need to be provided explicitly.
- **Reliability**—An Incomplete Abstraction may not implement the required functionality, resulting in defects. Furthermore, when methods are missing, the clients may have to work around to provide the required functionality, which can be error-prone and lead to runtime problems. For instance, when a method for locking is provided in an abstraction without the related unlock method, the clients will not be able to release the lock. When clients try to implement the unlock functionality in their code, they may try to directly access the internal data structures of the abstraction; such direct access can lead to runtime errors.

3.3.6 ALIASES

This smell is also known in literature as:

- "Class supports incomplete behavior" [18]—This smell occurs when the public interface of a class is incomplete and does not support all the behavior needed by objects of that class.
- "Half-hearted operations" [63]—This smell occurs when interrelated methods provided in an incomplete or in an inconsistent way; this smell could lead to runtime problems.

3.3.7 PRACTICAL CONSIDERATIONS

Disallowing certain behavior

Sometimes, a designer may make a conscious design decision to not provide symmetric or matching methods. For example, in a read-only collection, only `add()` method may be provided without the corresponding `remove()` method. In such a case, the abstraction may appear incomplete, but is not a smell.

Using a single method instead of a method pair

Sometimes, APIs choose to replace symmetrical methods with a method that takes a boolean argument (for instance, to enforce a particular naming convention such as

JavaBeans naming convention that requires accessors to have prefixes "get," "is," or "set"). For example, classes such as `java.awt.MenuItem` and `java.awt.Component` originally supported `disable()` and `enable()` methods. These methods were deprecated and are now replaced with `setEnabled(boolean)` method. Similarly, `java.awt.Component` has the method `setVisible(boolean)` that deprecates the methods `show()` and `hide()`. One would be tempted to mark these classes as Incomplete Abstractions since they lack symmetric methods, i.e., `getEnabled()` and `getVisible()` respectively. However, since there is no need for corresponding getter methods (as these methods take a boolean argument), these classes do not have Incomplete Abstraction smell.

3.4 MULTIFACETED ABSTRACTION

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

3.4.1 RATIONALE

An important enabling technique to effectively apply the principle of abstraction is to *assign single and meaningful responsibility* for each abstraction. In particular, the Single Responsibility Principle says that an abstraction should have a single well-defined responsibility and that responsibility should be entirely encapsulated within that abstraction. An abstraction that is suffering from Multifaced Abstraction has more than one responsibility assigned to it, and hence violates the principle of abstraction.

The realization of multiple responsibilities within a single abstraction leads to a low degree of cohesion among the methods of the abstraction. Ideally, these responsibilities should have been separated out into well-defined, distinct, and cohesive abstractions. Thus, the low degree of cohesion within a Multifaceted Abstraction also leads to the violation of the principle of modularization.

Furthermore, when an abstraction includes multiple responsibilities, it implies that the abstraction will be affected and needs to be changed for multiple reasons. Our experience shows that there is often a strong correlation between the frequency of change in elements of a design and the number of defects in that element. This means that a Multifaceted Abstraction may likely suffer from a greater number of defects. A Multifaceted Abstraction lowers the quality of the design and should be refactored to avoid building up the technical debt. Since the abstraction has multiple "faces" or "responsibilities", it is named Multifaceted Abstraction.

3.4.2 POTENTIAL CAUSES

General-purpose abstractions

When designers introduce an abstraction with a generic name (examples: `Node`, `Component`, `Element`, and `Item`), it often becomes a "placeholder" for providing all the functionality *related* (but not necessarily belonging) to that abstraction. Hence, general purpose abstractions often exhibit this smell.

Evolution without periodic refactoring
When a class undergoes extensive changes over a long period of time without refactoring, other responsibilities start getting introduced in these classes and design decay starts. In this way, negligence toward refactoring leads to the creation of monolithic blobs that exhibit multiple responsibilities.

The burden of processes
Sometimes the viscosity of the software and environment (discussed earlier in Section 2.2.5) serves to discourage the adoption of good practices. As discussed in the case of Insufficient Modularization smell (Section 5.2), to circumvent the long process that must be followed when a new class is added to the design, developers may choose to integrate new unrelated features within existing classes leading to Multifaceted Abstraction smell.

Mixing up concerns
When designers don’t give sufficient attention to the separation of different concerns (e.g., not separating domain logic from presentation logic), the resulting abstraction will have Multifaceted Abstraction smell.

3.4.3 EXAMPLE
In his book, Neal Ford mentions `java.util.Calendar` class as an example of a class having multiple responsibilities [64]. A class abstracting real-world calendar functionality is expected to support dates, but the `java.util.Calendar` class supports time related functionality as well, and hence this class suffers from Multifaceted Abstraction smell.

Since methods supporting date and time are combined together, the interface of `Calendar` class is large and difficult to comprehend. This difficulty in using this class has prompted the creation of alternatives. One such alternative is Joda,[2] which is a replacement of standard Java date and time API. In fact, due to the difficulties with the existing `Calendar` class and other classes related to date and time processing, a new date and time API has been introduced in JDK version 8.

Note: Classes exhibiting Multifaceted Abstraction are usually large and complex. For example, `java.util.Calendar` class (in JDK 7) spans 2825 lines of code and has 67 methods and 71 fields! However, large or complex implementation is not an essential characteristic of classes having Multifaceted Abstraction smell; see the discussion on Insufficient Modularization smell (Section 5.2).

[2] http://www.joda.org/joda-time/.

ANECDOTE

While working as an independent consultant, one of the authors was asked by a start-up company to help them identify refactoring candidates in a project that involved graphical displays in hand-held devices. The author ran many tools including object-oriented metrics, design smells detection, and clone detector tools on the code-base to identify refactoring candidates. One of the classes identified for refactoring was a class named `Image`. The class had 120 public methods realizing multiple responsibilities in more than 50,000 lines of code. When running a design smell detection tool, the tool showed Image class to have “Schizophrenic class” smell (the tool defined “Schizophrenic class” smell as a class having a large public interface with clients using disjoint group of methods from the class interface).

When the author proposed refactoring of the `Image` class, the team objected, saying that the `Image` class followed the SRP and hence there was no need to refactor it. Specifically, they argued that the class had a comprehensive set of methods that related to loading, rendering, manipulating, and storing the image, and all these methods were logically related to images.

In their argument, the project team completely missed the notion of “granularity” of responsibilities. If you take a look at an image processing application, almost everything is related to images: does it mean we should put everything in a single class named `Image`? Absolutely not. Each abstraction has a concrete, specific, and precise responsibility (instead of a coarse-grain high-level responsibility).

Coming back to the case under discussion, upon further probing, the team lead admitted that the class was one of the most frequently modified classes during defect fixes and feature enhancements. He also added that they changed the `Image` class for different reasons. The author and the team lead then had several discussions on how this problem could be avoided. Finally, the team lead accepted that there was an urgent need to refactor the `Image` class. As a result, four modules (or namespaces)—named `ImageLoader`, `ImageRenderer`, `ImageProcessor`, and `ImageWriter` (relating to loading, rendering, manipulating, and persisting functionality, respectively)—were extracted from the original `Image` class. With this refactoring, the main benefit was that the changes became localized. For example, if any change that related to image rendering was required, only the code in `ImageRenderer` module would need to be changed.

The main take-away that can be identified from this experience is that “responsibility” (as in Single “Responsibility” Principle) does not mean logical grouping of all functionality related to a concept. Rather “responsibility” refers to a concrete, specific, and precise responsibility that has one reason to change!

3.4.4 SUGGESTED REFACTORING

When a class has multiple responsibilities, it will be non-cohesive. However, it is likely that you will find methods and fields that form logical clusters. These logical clusters are candidates for applying “extract class” refactoring. In particular, new class(es) can be created out of existing classes having Multifaceted Abstraction smell and the relevant methods and fields can be moved to the new class(es).

Note that, depending on the context, the extracted class may either be used directly only by the original class, or may be used by the clients of the original class (Figure 3.11).

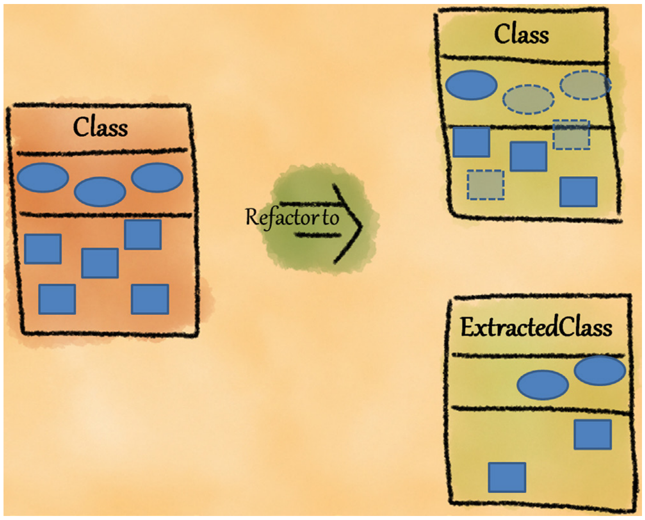

FIGURE 3.11

‘Extract class’ refactoring for Multifaceted Abstraction.

Suggested refactoring for the Example

For the `Calendar` class, a possible refactoring is to extract time-related functionality from the `Calendar` class into a new `Time` class and move the relevant methods and fields into the newly extracted class. Java 8 has introduced new classes supporting date and time (and other classes such as clocks, duration, etc.) in a package named `java.time` so that future clients can use this new package instead.

3.4.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—A Multifaceted Abstraction increases the cognitive load due to multiple aspects realized into the abstraction. This impacts the understandability of the abstraction.
- **Changeability** and **Extensibility**—When an abstraction has multiple responsibilities, it is often difficult to figure out what all members within the abstraction need to be modified to address a change or an enhancement. Furthermore, a modification to a responsibility may impact unrelated responsibilities provided within the same abstraction, which may lead to ripple effects across the entire design.
- **Reusability**—Ideally, a well-formed abstraction that performs a single responsibility has the potential to be reused as a unit. 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 impacting the abstraction's reusability. Furthermore, 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 affecting its reusability.

- **Testability**—When an abstraction has multiple responsibilities, these responsibilities may be entwined with each other, making it difficult to test each responsibility separately. Thus, testability is impacted.
- **Reliability**—The effects of modification to an abstraction with intertwined responsibilities may be unpredictable and lead to runtime problems.

3.4.6 ALIASES

This smell is also known in literature as:

- "Divergent change" [7]—This smell occurs when a class is changed for different reasons.
- "Conceptualization abuse" [30]—This smell occurs when two or more non-cohesive concepts have been packed into a single class of the system.
- "Large class" [7,24,57,58]—This smell occurs when a class has "too many" responsibilities.
- "Lack of cohesion" [59]—This smell occurs when there is a large type in a design with low cohesion, i.e., a "kitchen sink" type that represents many abstractions.

3.4.7 PRACTICAL CONSIDERATIONS

None.

3.5 UNNECESSARY ABSTRACTION

This smell occurs when an abstraction that is actually not needed (and thus could have been avoided) gets introduced in a software design.

3.5.1 RATIONALE

A key enabling technique to apply the principle of abstraction is to assign single and meaningful responsibility to entities. However, when abstractions are created unnecessarily or for mere convenience, they have trivial or no responsibility assigned to them, and hence violate the principle of abstraction. Since the abstraction is needlessly introduced in the design, this smell is named Unnecessary Abstraction.

3.5.2 POTENTIAL CAUSES

Procedural thinking in object-oriented languages

According to Johnson et al. [60], programmers from procedural background who are new to object-oriented languages tend to produce classes that “do” things instead of “being” things. Classes in such designs tend to have just one or two methods with data located in separate “data classes.” Such classes created from procedural thinking usually do not have unique and meaningful responsibilities assigned to them.

Using inappropriate language features for convenience

Often, programmers introduce abstractions that were not originally intended in design for “getting-the-work-done” or just for “convenience.” Using “constant interfaces”[3] instead of enums, for example, is convenient for programmers. By implementing constant interfaces in classes, programmers don’t have to explicitly use the type name to access the members and can directly access them in the classes. It results in unnecessary interfaces or classes that just serve as placeholders for constants without any behavior associated with the classes.

Over-engineering

Sometimes, this smell can occur when a design is over-engineered. Consider the example of a customer ID associated with a `Customer` object in a financial application. It may be overkill to create a class named `CustomerID` because the `CustomerID` object would merely serve as holder of data and will not have any non-trivial or meaningful behavior associated with it. A better design choice in this case would be to use a string to store the customer ID within a `Customer` object.

3.5.3 EXAMPLES

Example 1

Consider the `java.util.FormattableFlags` class from JDK 7 that holds three flag values used in format specifiers; namely, “-,” “S,” and “#.” These flags are encoded as integer values 1, 2, and 4 (respectively). Interestingly, this class was introduced in Java 1.5, which also saw the introduction of enumerations into the language. Since an enumeration could have been used instead of the public static int fields, the `java.util.FormattableFlags` class becomes unnecessary.

[3] Using the interface feature in Java as holder for constant values.

```
public class FormattableFlags {
  // Explicit instantiation of this class is prohibited.
  private FormattableFlags() {}
  /** Left-justifies the output. */
  public static final int LEFT_JUSTIFY = 1<<0;//'-'
  /** Converts the output to upper case */
  public static final int UPPERCASE = 1<<1;//'S'
  /** Requires the output to use an alternate form. */
  public static final int ALTERNATE = 1<<2;//'#'
}
```

Example 2

Consider the `javax.swing.WindowConstants` interface that defines four constants that are used to control the window-closing operations. Here is the code (documentation comments edited to conserve space):

```
public interface WindowConstants {
  /** The do-nothing default window close operation. */
  public static final int DO_NOTHING_ON_CLOSE = 0;
  /** The hide-window default window close operation */
  public static final int HIDE_ON_CLOSE = 1;
  /** The dispose-window default window close operation. */
  public static final int DISPOSE_ON_CLOSE = 2;
  /** The exit application default window close operation. */
  public static final int EXIT_ON_CLOSE = 3;
}
```

This interface is an example of "constant interface" and is a poor use of the interface feature provided by the language. Why would developers or designers use an interface to hold constants? First, enumerations were only introduced in Java 1.5, so developers had to use existing features. Second, it was convenient for classes to use inheritance rather than delegation to use constants defined in the interfaces. The reason is that the classes that implement that interface could conveniently access the constants defined in the interface without explicitly qualifying the constant values with the name of the interface! A natural question is: Why use interfaces to hold constants instead of using classes? If a class were to be used for defining constants, then its derived classes cannot extend any other class (note that Java does not support multiple class inheritance); hence, interfaces were preferred over classes for holding constants.

Having understood constant interfaces, note that there are classes such as `JFrame`, `JInternalFrame`, and `JDialog` that inherit the `WindowConstants` interface. Another such example of constant interface use is `java.io.ObjectStreamConstants` that is implemented by `ObjectInputStream` and `ObjectOutputStream` classes.

The approach of using a constant interface and implementing it suffers from the following problems:

- The derived classes are "polluted" with constants that may not be relevant to that derived class.
- These constants are implementation details and exposing them through an interface violates encapsulation.
- When constants are part of an interface, changes to the constants can break the existing clients.

To summarize, an interface serves as a protocol that the implementing classes must support. Defining an interface to use it as a holder of constants is an abuse of the abstraction mechanism.

Example 3

Consider the case of an e-commerce application that has two classes: namely, `BestSellerBook` and `Book`. Whenever the client wants to create a best-seller book, it creates an instance of a `BestSellerBook`. Internally, `BestSellerBook` delegates all the method calls to the `Book` class and does nothing else. Clearly, the `BestSellerBook` abstraction is unnecessary since its behavior is exactly the same as the `Book` abstraction.

3.5.4 SUGGESTED REFACTORING

The generic refactoring suggestions to address this smell include the following:

- As Fowler suggests [7], "a class that isn't doing enough to pay for itself should be eliminated."
- Consider applying "inline class" refactoring [7] to merge the class with another class.
- If a class or interface is being introduced to encode constants, check if you can use a more suitable alternative language feature such as enumerations instead.

Suggested refactoring for Example 1

The suggested refactoring is to make the `FormattableFlags` an enumeration and use it with `java.util.Formatter` class, which provides support for formatted output.

Suggested refactoring for Example 2

The suggested refactoring is to make the `WindowsConstants` an enumeration. With this refactoring, classes such as `JFrame`, `JInternalFrame`, and `JDialog` can use this enumeration instead. A similar refactoring applies to the `ObjectStreamConstants` example as well.

Suggested refactoring for Example 3

For the case of the e-commerce application that has two classes; namely, `BestSellerBook` and `Book`, there are many possible refactoring solutions. One solution, for instance, is to remove the `BestSellerBook` class and instead add an attribute named `isBestSeller` (along with a getter and a setter) in the `Book` class. Now, when the client code wants to indicate if a book is a bestseller, it will set the attribute `isBestSeller` instead of creating an instance of the erstwhile `BestSellerBook` class.

3.5.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—Having needless abstractions in the design increases its complexity unnecessarily and affects the understandability of the overall design.
- **Reusability**—Abstractions are likely to be reusable when they have unique and well-defined responsibilities. An abstraction with trivial or no responsibility is less likely to be reused in a different context.

3.5.6 ALIASES

This smell is also known in literature as:

- "Irrelevant class" [51]—This smell occurs when a class does not have any meaningful behavior in the design.
- "Lazy class"/"Freeloader" [7,62]—This smell occurs when a class does "too little."
- "Small class" [57,60]—This smell occurs when a class has no (or too few) variables or no (or too few) methods in it.
- "Mini-class" [63]—This smell occurs when a public, non-nested class defines less than three methods and less than three attributes (including constants) in it.
- "No responsibility" [65]—This smell arises when a class has no responsibility associated with it.
- "Agent classes" [51]—This smell arises when a class serve as an "agent" (i.e., they only pass messages from one class to another), indicating that the class may be unnecessary.

3.5.7 PRACTICAL CONSIDERATIONS

Delegating abstractions in design patterns

Some design patterns (e.g., Mediator, Proxy, Façade, and Adapter) that employ delegation have a class that may appear to be an Unnecessary Abstraction. For example, in case of the Object Adapter pattern, the `Adapter` class may appear to merely delegate client requests to the appropriate method on the `Adaptee` [54]. However, the primary intention behind the `Adapter` class is to fulfill the specific, well-defined responsibility of adapting the `Adaptee's` interface to the client needs. Hence, one has to carefully consider the context before deciding whether an abstraction that just performs delegation is unnecessary or not.

Accommodating variations

Consider the example of `java.lang.Math` and `java.lang.StrictMath` classes, which provide almost similar math-related functionality. The JavaDoc for `StrictMath` notes: "By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation." It may appear from the JavaDoc description that `Math` is an Unnecessary Abstraction that simply delegates calls to `StrictMath`. However, although both support math-related functionality, `StrictMath` methods return exactly the same results irrespective of the platform because the implementation conforms to the relevant floating-point standard, whereas `Math` methods may use native hardware support for floating-point numbers and hence return slightly different results. Here, note that `Math` is less portable but can result in better performance when compared to `StrictMath`. Therefore, it is a conscious design decision to create two abstractions to accommodate variation in objectives, i.e., portability and performance.

3.6 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 derived abstractions

3.6.1 RATIONALE

One of the enabling techniques for applying the principle of abstraction is to assign a single and meaningful responsibility to an entity. When an abstraction is left unused in design, it does not serve a meaningful purpose in design, and hence violates the principle of abstraction.

Design should serve real needs and not imagined or speculative needs. Unrealized abstract classes and interfaces indicate unnecessary or speculative generalization, and hence are undesirable. This smell violates the principle YAGNI (You Aren't Gonna Need It), which recommends not adding functionality until deemed necessary [53]. Since the abstraction is left unutilized in the design, this smell is named Unutilized Abstraction.

3.6.2 POTENTIAL CAUSES

Speculative design

When designers attempt to make the design of a system "future-proof" or provide abstractions "just in case it is needed in future," it can lead to this smell.

Changing requirements

When requirements keep changing, the abstractions created for satisfying an earlier requirement often may not be needed anymore. However, when the abstraction continues to remain in the design, it becomes an Unutilized Abstraction.

Leftover garbage during maintenance

When maintenance or refactoring activities are performed without cleaning up the old abstractions, it could result in unreferenced abstractions.

Fear of breaking code

Often, one of the reasons why developers do not delete old code is that they are not sure if any other class in the code is still using it. This is especially true in large codebases where it is difficult to determine whether a piece of code is being used or not.

3.6.3 EXAMPLES

Example 1 (unreferenced abstractions)

This example is paraphrased from a bug report on unused classes in JDK.[4] The package `sun.misc` has classes that date back to early releases of JDK, and they were used by other classes in JDK internally. Later, many of the services provided by `sun.misc` package were provided as part of the public API. Eventually, the original clients of `sun.misc` package started using the services provided by the public API. Due to this, many of the original classes in `sun.misc` package became unreferenced abstractions. One such example is the internal class `sun.misc.Service` that was introduced in JDK 1.3, which was made redundant by the introduction of the class `java.util.ServiceLoader` in JDK version 1.6 (which is part of the public API). Hence, the original `sun.misc.Service` is an Unutilized Abstraction.

Example 2 (unreferenced abstractions)

In a large Java project that one of the authors was involved in, there was extensive use of concurrency. The code was developed using JDK 1.3 and contained several concurrent utilities such as a concurrent version of `HashMap`, a FIFO (First-In-First-Out) buffer that would block, and an array list implementation that used copy-on-write approach. When JDK 1.5 introduced `java.util.concurrent` package, developers started using the classes in this package without removing the original concurrent utilities which became Unutilized Abstractions.

Example 3 (orphan abstractions)

One of the authors was involved in the development of a visualization tool integrated into an Integrated Development Environment (IDE). The tool would

[4] See the original bug report "JDK-6852936: Remove unused classes from sun.misc" here: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6852936.

visually show the data flow of potential bugs such as null-pointer access, divide-by-zero, etc. While developing the tool, the designer thought that it would be a useful feature to generate reports of the tool results. Since he planned to create PDF and HTML versions of the reports, he created the class hierarchy shown in Figure 3.12. All three were abstract classes and they did not have much code in it, because they were "placeholders" for future implementation. However, as the primary objective of the tool was to visualize data flow so that developers can view how a control-flow path can result in a defect, report generation was not of much importance. So, report generation, although likely to be a useful feature, was never implemented. For this reason, these three classes (shown in Figure 3.12) are Unutilized Abstractions. Note that the hierarchy (in Figure 3.12) may appear to suffer from Unnecessary Hierarchy (Section 6.2). However, in the case of Unnecessary Hierarchy the types in the hierarchy are utilized, which is not the case here.

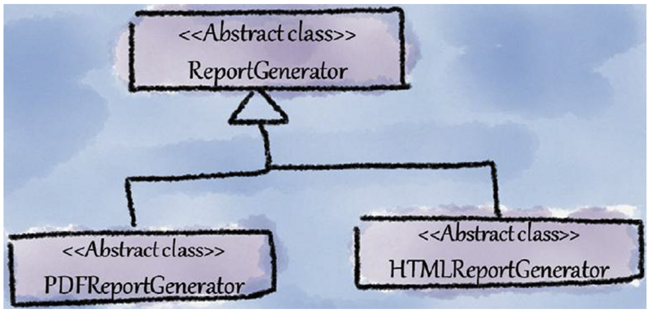

FIGURE 3.12

Unutilized report generation classes (Example 3).

ANECDOTE

During maintenance and refactoring, developers often refactor code bases that result in some of the old classes becoming obsolete. However, such code segments are often left in the code base (sometimes by mistake), making the source code bulge in size. Consider this documentation from the Java bug database for a set of classes in JDK source base (Check the original bug report "JDK-7161105: unused classes in jdk7u repository" here: http://bugs.sun.com/view_bug.do?bug_id=7161105):

"The ObjectFactory/SecuritySupport classes that were duplicated in many packages were replaced by a unified set under internal/utils. When the jaxp 1.4.5 source bundle was dropped, however, it was a copy of the source bundle on top of the original source, therefore leaving these classes behind unremoved."

The left-over code was later removed from the source base. What is the lesson that we can learn from this bug report? During refactoring or maintenance, it is important to ensure that old code is deleted from the code base to avoid its bloating due to dead code and unused classes.

3.6.4 SUGGESTED REFACTORING

The simplest refactoring is to remove the Unutilized Abstraction from the design. However, in the case of APIs, removing an abstraction is not feasible since there may be client code (or legacy code) that may be still referring to it. In these cases, such abstractions may be marked “obsolete” or “deprecated” to explicitly state that they must not be used by new clients.

Suggested refactoring for Example 1

All the uses of the `sun.misc.Service` class can be replaced by the use of `java.util.ServiceLoader` class. Therefore, for all practical purposes, there is no need for `sun.misc.Service` class and hence the suggested refactoring is to remove it from the code base. In fact, `sun.misc.Service` has been removed from JDK source code and JDK 9 will not have it.[5]

Suggested refactoring for Example 2

The following refactoring suggestions can be made for Example 2 considering the mapping between the project-specific concurrent utilities and the standard utilities introduced in Java 1.5 and later versions:

- The concurrent version of `HashMap` can be replaced with the use of `java.util.concurrent.ConcurrentHashMap`
- The blocking FIFO buffer can be replaced with the use of `java.util.concurrent.LinkedBlockingQueue`
- The array list implementation that used the copy-on-write approach could be replaced with the use of `java.util.concurrent.CopyOnWriteArrayList`

Note that performing the refactoring to make such replacements would require extensive code/design analysis and reviews in addition to extensive testing. This is because such replacements can result in subtle concurrency bugs that are hard to find, reproduce, or test.

Suggested refactoring for Example 3

Since report generation is a feature that is not needed and the hierarchy was added based on a speculated need, the suggested refactoring is to remove the entire hierarchy from the design.

[5] See the closed bug report ‘JDK-8034776: Remove sun.misc.Service’ in: http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8034776.

ANECDOTE

Unutilized Abstraction is a commonly occurring smell across development organizations. Interestingly, the main reason behind the occurrence of this smell is changing requirements. In one instance that the authors are aware of, the project was following the Scrum process. In each sprint, suitable tasks would be identified from the backlog for development. Just like in other real-world projects, after a certain sprint, the customer changed his requirements, and consequently a new feature was expected to replace the old one.

Before the next sprint commenced, the allocation of tasks for the development of the new feature was done. It turned out that the new task was assigned to a different developer. This developer, instead of reusing the existing implementation, overlooked it and reinvented the wheel. In other words, he created new classes that were doing the same thing but the old classes were never removed and were left unused leading to the Unutilized Abstraction smell.

3.6.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—Presence of unused abstractions in design pollutes the design space and increases cognitive load. This impacts understandability.
- **Reliability**—The presence of unused abstractions can sometimes lead to runtime problems. For instance, when code in unused abstractions gets accidentally invoked, it can result in subtle bugs affecting the reliability of the software.

3.6.6 ALIASES

This smell is also known in literature as:

- "Unused classes" [35,55]—This smell occurs when a class has no direct references and when no calls to that class's constructor are present in the code.
- "Speculative generality" [7]—This smell occurs when classes are introduced, speculating that they may be required sometime in future.

3.6.7 PRACTICAL CONSIDERATIONS

Unutilized Abstractions in APIs

Class libraries and frameworks usually provide extension points in the form of abstract classes or interfaces. They may appear to be unused within the library or framework. However, since they are extension points that are intended to be used by clients, they cannot be considered as Unutilized Abstractions.

3.7 DUPLICATE ABSTRACTION

This smell arises when two or more abstractions have identical names or identical implementation or both. The design smell exists in two forms:

- **Identical name**—This is when the names of two or more abstractions are identical. While two abstractions can accidentally have the same name, it needs to be analyzed whether they share similar behavior. If their underlying behavior is not related, the main concern is simply the identical names. This would impact understandability since two distinct abstractions have an identical name. However, in case the underlying behavior of the two abstractions is also identical, it is a more serious concern and could indicate the "identical implementation" form of Duplicate Abstraction.
- **Identical implementation**—This is when two or more abstractions have semantically identical member definitions; however, the common elements in those implementations have not been captured and utilized in the design. Note that the methods in these abstractions might have similar implementation, but their signatures might be different. Additionally, identical implementation occurring in siblings within an inheritance hierarchy may point to an Unfactored Hierarchy smell (see Section 6.3).

This design smell also includes the combination of these forms. We have come across designs with "duplicate" abstractions that have identical names, identical method declarations (possibly with different signatures), and similar implementation.

Identical implementation could occur in four forms following the nature of "code clones" (fragments of code that are very similar). The sidebar "types of code clones" summarizes these four forms.

TYPES OF CODE CLONES

Code clones can be similar textually in the code, or they could be similar in their functionally but could differ textually. Based on the level of similarity, code clones can be considered to be of four types [16]:

Textually Similar Clones

Type 1: Two code fragments are clones of type-1 when the fragments are exactly identical except for variations in whitespace, layout, and comments.
Type 2: Two code fragments are clones of type-2 when the fragments are syntactically identical except for variation in symbol names, whitespace, layout, and comments.
Type 3: Two code fragments are clones of type-3 when the fragments are identical except some statements changed, added, or removed, in addition to variation in symbol names, whitespace, layout, and comments.

Functionally Similar Clones

Type 4: Two code fragments are clones of type-4 when the fragments are semantically identical but implemented by syntactic variants.

3.7.1 RATIONALE

Avoid duplication is an important enabling technique for the effective application of the principle of abstraction.

If two or more abstractions have an identical name, it affects understandability of the design. Developers of client code will be confused and unclear about the choice of the abstraction that should be used by their code.

If two or more abstractions have identical implementation (i.e., they have duplicate code), it becomes difficult to maintain them. Often, a change in the implementation of one of these abstractions will need to be reflected across all other duplicates. This introduces not only an overhead but also the possibility of subtle difficult-to-trace bugs. Duplication should be avoided as much as possible to reduce the extent of change required.

In summary, this smell indicates a violation of the DRY (Don't Repeat Yourself) principle. The DRY principle mandates that every piece of knowledge must have a single unambiguous representation within a system. If the DRY principle is not followed, a modification of an element within the system requires modifications to other logically unrelated elements making maintainability a nightmare. Since there is duplication among abstractions in the design, this smell is named Duplicate Abstraction.

3.7.2 POTENTIAL CAUSES

A Duplicate Abstraction smell can arise due to many reasons. While some of these reasons are generic, some are specific to a particular programming paradigm or platform.

Copy-paste programming

The "get-the-work-done" mindset of a programmer leads him to copy and paste code instead of applying proper abstraction.

Ad hoc maintenance

When the software undergoes haphazard fixes or enhancements over many years, it leaves "crufts"[6] with lots of redundant code in it.

Lack of communication

Often, in industrial software, code duplication occurs because different people work on the same code at different times in the life cycle of the software. They are not aware of existing classes or methods and end up re-inventing the wheel.

Classes declared non-extensible

When a class that needs to be extended is declared non-extensible (for example, by declaring a class as final in Java), developers often resort to copying the entire code in a class to create a modified version of the class.

[6]Wikipedia: Cruft is jargon for anything that is left over, redundant and getting in the way. It is used particularly for superseded and unemployed technical and electronic hardware and useless, superfluous or dysfunctional elements in computer software.

3.7.3 EXAMPLES

JDK has numerous classes with identical names. Specifically, of the 4005 types in Java 7, 135 type names are duplicates, which is a substantial 3.3%! Let us discuss a few of these Duplicate Abstractions.

Example 1

There are three different classes with the name `Timer` in JDK. The description of these classes as per JDK documentation is as follows:

- `javax.swing.Timer` is meant for executing objects of type `ActionEvent` at specified intervals.
- `java.util.Timer` is meant for scheduling a thread to execute in the future as a background thread.
- `javax.management.timer.Timer` is meant for sending out an alarm to wake up the listeners who have registered to get timer notifications.

Since all three timers have similar descriptions (they execute something in future after a given time interval), choosing the right `Timer` class can be challenging to inexperienced programmers.

Example 2

Classes `java.util.Date` and its derived class `java.sql.Date` share the same name (see Figure 3.13)! The compiler does not complain that the base and derived classes have the same name because these classes belong to different packages. However, it is very confusing for the users of these classes. For example, when both these classes are imported in a program, it will result in a name ambiguity that must be resolved manually by explicitly qualifying the class names.

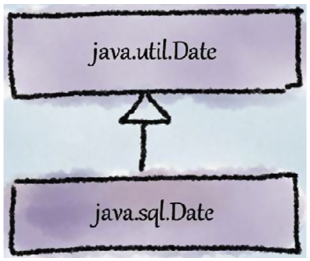

FIGURE 3.13

Classes with identical name `Date' in JDK (Example 2).

Example 3

The `Scanner` class of `org.eclipse.jdt.internal.compiler.parser` package is almost a complete duplication of the `PublicScanner` class in `org.eclipse.jdt.`

core.compiler package. About 3500 lines of code are identical across these two classes. The JavaDoc description of the Scanner class says:

"Internal Scanner implementation. It is mirrored in org.eclipse.jdt.core.compiler public package where it is API."

Both classes share identical public method signatures as well as implementation, and hence suffer from the "identical implementation" form of Duplicate Abstraction.

ANECDOTE

An very unusual problem was reported by one of the attendees during one of our guest lectures on design smells. The attendee was involved as a consultant architect in a large Java project. He found that the project team had spent several hours debugging the following problem.

While invoking a method from a third-party library, an exception named Missing-ResourceException was getting thrown at times. To handle that exception, one of the team members had added a catch handler for that exception and logged that exception using a logging library. However, what was frustrating to the team was that even after adding the catch handler, the application sometimes crashed with the same exception!

To aid debugging, the attendee suggested that the logging code be removed and the stack trace be printed on the console. This was done and the application was executed repeatedly. Eventually, when the crash occurred again, it was found that the exception thrown was not java.util.MissingResourceException which the code was handing, but was a custom exception named MissingResourceException that was defined in a package in the third-party library. Since these exceptions are RuntimeExceptions, the compiler did not issue any error for the catch block for catching an exception that could never get thrown from that code block!

Although this appeared to be a programming mistake, a deeper reflection revealed that it was in fact a design problem: the design specification did not mention specifically the type of exception that should have been handled. This, in combination with the identical names of the two exception classes resulted in the programmer handling the wrong exception.

This incident also highlighted how careful one should be when designing an API. In this case, since the name of an exception defined by the third-party API clashes with an existing JDK exception, it resulted in problems for the clients of the third-party API.

In summary, the important lessons that can be learned from this experience are:

- Design problems can cause hard-to-find defects
- Naming is important; avoiding duplicate names is especially important!

3.7.4 SUGGESTED REFACTORING

For identical name form, the suggested refactoring is to rename one of the abstractions to a unique name.

In the case of the identical implementation form of Duplicate Abstraction, if the implementations are exactly the same, one of the implementations can be removed. If the implementations are slightly different, then the common implementation in the duplicate abstractions can be factored out into a common class. This could be an existing supertype in the existing hierarchy (see refactoring for Unfactored Hierarchy (Section 6.3)) or an existing/new class which can be "referred to" or "used" by the duplicate abstractions.

Suggested refactoring for Example 1

For the `Timer` class example, since the main concern is identical name, the refactoring suggestion is to change their names so that they are unique. `javax.swing.Timer` is meant for executing objects of type `ActionEvent`, so it can be renamed as `ExecutionTimer` or `EventTimer`. Similarly, the `Timer` implementation from `java.util` can be renamed to `AlarmTimer`. The `java.util.Timer` can be retained as it is.

Suggested refactoring for Example 2

For the `Date` class example, let us take a look at the JavaDoc description for `java.sql.Date`.

> "To conform with the definition of SQL DATE, the millisecond values wrapped by a java.sql.Date instance must be 'normalized' by setting the hours, minutes, seconds, and milliseconds to zero in the particular time zone with which the instance is associated."

Following the hints from the words "wrapped by" in the JavaDoc description, a better design could be to "wrap" a `java.util.Date` instance in `java.sql.Date`, i.e., convert inheritance to delegation (see Broken Hierarchy in Section 6.8 for further details). Further since `java.sql.Date` conforms to SQL DATE, it is preferable to rename it as `java.sql.SQLDate` which will clearly differentiate it from the plain class name `java.util.Date` (see Figure 3.14).

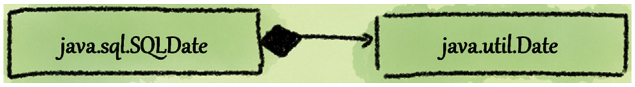

FIGURE 3.14

Suggested refactoring for java.util.Date and java.sql.Date classes (Example 2).

Suggested refactoring for Example 3

With respect to the `Scanner` example, the classes `Scanner` and `PublicScanner` suffer from a high degree of identical implementation. A possible refactoring is to use inheritance but this would be appropriate only if the duplicate percentage is moderate and both the abstractions vary quite a bit. However, since the percentage of duplication is 99.9, a more appropriate solution is to delete one of the classes and make the other one compatible with the clients of the deleted class.

3.7.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—Developers can become confused about which abstraction to use when there are two or more abstractions with identical names or implementation. Further, duplicate implementations bloat the code. These factors impact the understandability of the design.
- **Changeability** and **Extensibility**—Change or enhancement involving one abstraction potentially requires making the same modification in the duplicate abstractions as well. Hence, changeability and extensibility are considerably impacted.
- **Reusability**—Duplicate abstractions often have slightly different implementations (especially Type 3 and Type 4 clones). The differences in implementations are usually due to the presence of context-specific elements embedded in the code. This makes the abstractions hard to reuse in other contexts; hence, reusability of the abstractions is impacted.
- **Reliability**—When two abstractions have identical names, a confused developer may end-up using the wrong abstraction. For example, he may type cast to a wrong type, leading to a runtime problem. In case of abstractions with identical implementations, a modification in one of the abstractions needs to be duplicated across the other copies failing which a defect may occur.

3.7.6 ALIASES

This smell is also known in literature as:

- "Alternative classes with different interfaces" [7]—This smell occurs when classes do similar things, but have different names.
- "Duplicate design artifacts" [74]—This smell occurs when equivalent design artifacts are replicated throughout the architecture.

3.7.7 PRACTICAL CONSIDERATIONS

Accommodating variations

One reason why duplicate abstractions may exist is to support synchronized and unsynchronized variants. A synchronized variant of an abstraction may have used synchronization constructs heavily and this may lead to creating separate abstractions (that suffer from Duplicate Abstraction smell) corresponding to the two variants. An example of this is seen in `java.util.Vector` and `java.util.ArrayList` classes that have similar method definitions. The main difference between these classes is that the former is thread-safe and the latter is not thread-safe.

Duplicate type names in different contexts

It is hard to analyze and model large domains and create a unified domain model. In fact, “total unification of the domain model for a large system will not be feasible or cost-effective” [17]. One solution offered by Domain Driven Design is to divide the large system into “Bounded Contexts.” In this approach, the resulting models in different contexts may result in types with same names. Since Bounded Context is one of the patterns that help deal with the larger problem of modeling large domains, such types with same names in different contexts is acceptable.

Lack of language support for avoiding duplication

Many methods and classes are duplicated in JDK because generics support is not available for primitive types. For example, the code for methods such as `binarySe-arch`, `sort`, etc. are duplicated seven times in the `java.util.Arrays` class because it is not possible to write a single generic method that takes different primitive type arrays. This results in the class suffering from a bloated interface.