CHAPTER

Encapsulation Smells

4

The principle of encapsulation advocates separation of concerns and information hiding through techniques such as hiding implementation details of abstractions and hiding variations.

Consider the example of a car. Do you need to know exactly how the engine of the car works in order for you to be able to drive it? Is it really required for you to know how the anti-braking system (ABS) of your car works? Well, you may know these details (if you are an automobile engineer), but these details are not required for you to drive the car (Figure 4.1).

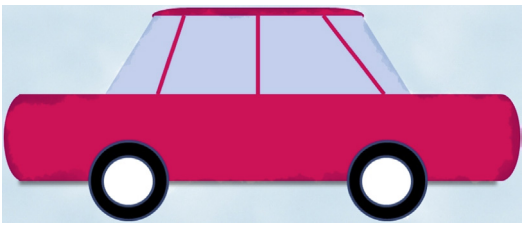

FIGURE 4.1

A car hides internal details (e.g., engine) from the its users.

This is precisely what the principle of Encapsulation does—it hides details that are not really required for the user of the abstraction (the abstraction in this case is the car). In addition, the principle of Encapsulation helps an abstraction to hide variation in the implementation details. For instance, whether your car has a petrol engine or a diesel engine, it does not change the way you drive your car.

The principle of encapsulation complements the principle of abstraction through information hiding. Encapsulation disallows (or "hides") the users of the abstraction from seeing internal details of the abstraction. In addition, encapsulation hides variation in implementation; the idea of hiding variation is explicitly stated by Gamma et al. [54]: "Encapsulate what varies." Hence, there are two techniques that enable effective application of the principle of encapsulation (see Figure 4.2):

- **Hide implementation details**. An abstraction exposes to its clients only "what the abstraction offers" and hides "how it is implemented." The latter,

Refactoring for Software Design Smells. http://dx.doi.org/10.1016/B978-0-12-801397-7.00004-7

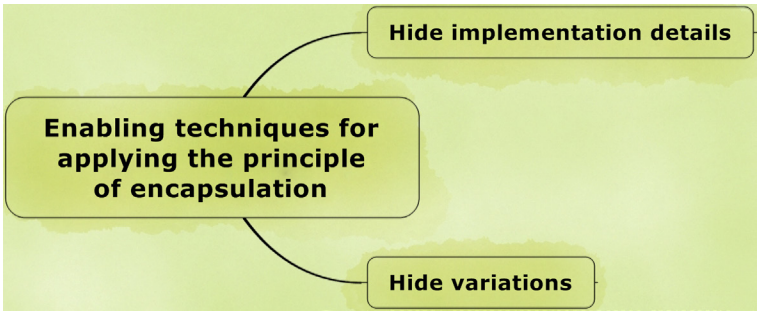

FIGURE 4.2

Enabling techniques for the principle of encapsulation.

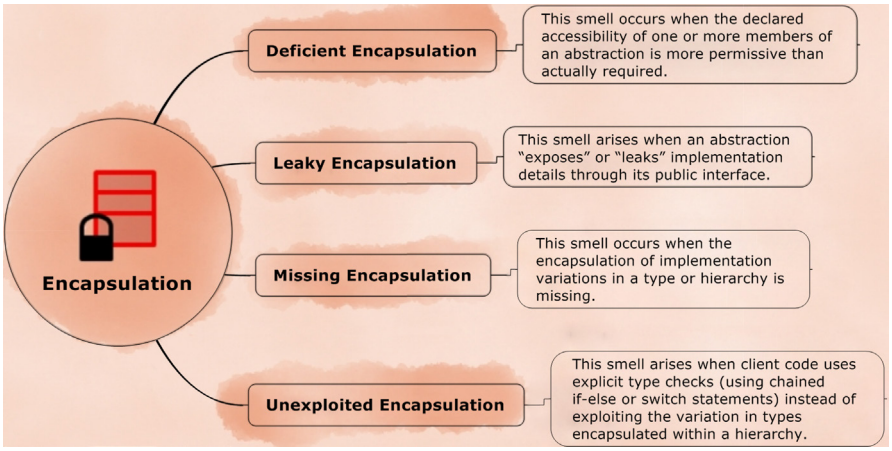

FIGURE 4.3

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

that is, the details of the implementation include the internal representation of the abstraction (e.g., data members and data structures that the abstraction uses) and details of how the method is implemented (e.g., algorithms that the method uses).

- **Hide variations**. Hide implementation variations in types or hierarchies. With such variations hidden, it is easier to make changes to the abstraction's implementation without much impact to the clients of the abstraction.

All of the smells described in this chapter map to a violation of one of the above enabling techniques. Figure 4.3 gives an overview of the smells that violate the principle of encapsulation, and Table 4.1 provides an overview of mapping between the design smells and the enabling technique(s) that they violate. The Rationale

Table 4.1 Design Smells and the Violated Encapsulation Enabling Techniques

Design Smells	Violated Enabling Technique
Deficient Encapsulation (Section 4.1), Leaky Encapsulation (Section 4.2)	Hide implementation details
Missing Encapsulation (Section 4.3), Unexploited Encapsulation (Section 4.4)	Hide variations

subsection in each of the smell descriptions provides a detailed explanation of how a particular smell violates an enabling technique.

In the rest of this chapter, we will discuss specific smells that result due to the violation of the principle of encapsulation.

4.1 DEFICIENT ENCAPSULATION

This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required. For example, a class that makes its fields public suffers from Deficient Encapsulation.

An extreme form of this smell occurs when there exists global state (in the form of global variables, global data structures, etc.) that is accessible to all abstractions throughout the software system.

4.1.1 RATIONALE

The primary intent behind the principle of encapsulation is to separate the interface and the implementation, which enables the two to change nearly independently. This separation of concerns allows the implementation details to be hidden from the clients who must depend only on the interface of the abstraction. If an abstraction exposes implementation details to the clients, it leads to undesirable coupling between the abstraction and its clients, which will impact the clients whenever the abstraction needs to change its implementation details. Providing more access than required can expose implementation details to the clients, thereby, violating the "principle of hiding."

Having global variables and data structures is a more severe problem than providing lenient access to data members. This is because global state can be accessed and modified by any abstraction in the system, and this creates secret channels of communication between two abstractions that do not directly depend on each other. Such globally accessible variables and data structures could severely impact the understandability, reliability, and testability of the software system.

Since this smell occurs when the internal state of an abstraction is inadequately hidden, we name this smell Deficient Encapsulation.

4.1.2 POTENTIAL CAUSES

Easier testability

Often developers make private methods of an abstraction public to make them testable. Since private methods concern implementation details of an abstraction, making them public compromises the encapsulation of the abstraction.

Procedural thinking in object oriented context

Sometimes, this smell occurs when developers from procedural programming background using an object oriented paradigm expose data as global variables when the data needs to be used by multiple abstractions.

Quick-fix solutions

Delivery pressures often force developers to take shortcuts (which happens to be one of the main causes of technical debt). For instance, in situations where the data is shared between only a couple of abstractions, developers find it easier to expose that data globally than to create an abstraction and pass it as a parameter to a method.

4.1.3 EXAMPLES

Example 1

A simple example of Deficient Encapsulation is from the `java.awt.Point` class. This class has public `x` and `y` fields in addition to public getter and setter methods for these fields (see Figure 4.4; only relevant members are shown in this figure). The class exhibits Deficient Encapsulation, since the accessibility of its members is more permissive (i.e., members are declared public) than actually required (i.e., they should have been declared private).

Interestingly, JDK has many examples of this smell in its implementation. In our analysis of JDK 7, we found 508 classes that have at least one public field, and hence

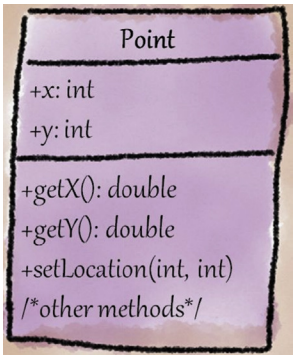

FIGURE 4.4

Public data members in java.awt.Point class (Example 1).

suffer from this smell. In fact, there are 96 classes that have five or more fields that are declared public. An extreme example is the class `com.sun.imageio.plugins.png.PNGMetadata` which has more than 85 public fields!

Example 2

Consider the class `SizeRequirements` from `java.swing.text` package in JDK 7. This class calculates information about the size and position of components that can be used by layout managers. Each `SizeRequirements` object contains information about either the width or height of a single component or a group of components.

Now consider the following public field in this class.

```
/** The alignment, specified as a value between 0.0 and 1.0,
inclusive. To specify centering, the alignment should be 0.5. */
public float alignment;
```

If the clients of `SizeRequirements` class were to directly access members like `alignment`, it would be difficult for the `SizeRequirements` class to ensure that the value of `alignment` remains valid as per the constraints specified in the comments (i.e., the value must be between 0.0 and 1.0).

CASE STUDY

It is common to observe rampant occurrences of Deficient Encapsulation smell in real-world projects. Although it is easy to understand why fields may be declared public, it is interesting to note in many real-world projects that methods that should have been declared private are instead made public.

When we questioned some development teams about this, most of them said that it was a conscious design choice: when methods are declared private, it is difficult to test them, so they are typically made public!

However, exposing all of the methods as public not only compromises design quality but can also cause serious defects. For instance, clients of the class will start directly depending on implementation details of the class. This direct dependency makes it difficult to change or extend the design. It also impacts the reliability, since clients can directly manipulate the internal details of the abstractions. Hence, ensuring the integrity of the internal state of the abstractions becomes difficult (if not impossible). In summary, making internal methods publicly accessible in the name of testability can have serious consequences.

So, how can we test classes without compromising their encapsulation?

There are many ways in which tests can be written without exposing methods as public. One way is to use reflection—tests can use reflection to dynamically load classes, examine their internals, and invoke methods (including private ones). Another way is to adopt a language such as Groovy that allows direct access to all the members (and ignores access specifiers) of a Java class.

Example 3

Consider the public classes `javax.swing.SwingUtilities`, `sun.swing.SwingUtilities2` and `sun.java.swing.SwingUtilities3` in Java 1.7. These utility classes provide services to the rest of the swing package. For instance,

`SwingUtilities3` class provides services for JavaFX applets. Since these classes are meant to be used only by the classes in swing package, they can be considered as "internal implementation details" of the swing package. In other words, the code in these utility classes is not meant to be used by the end-user applications.

In the context of Deficient Encapsulation, consider the documentation provided within `SwingUtilities2` and `SwingUtilities3` classes.

```
/**
* A collection of utility methods for Swing.
* <p>
* WARNING: While this class is public, it should not be treated as.
* public API and its API may change in incompatible ways between
* dot dot releases and even patch releases.
* You should not rely on this class even existing.
*/
```

Since these classes are declared public, they are accessible to client code that can directly use them. Such direct use of implementation-level methods of the swing package violates the principle of encapsulation, and hence these classes suffer from Deficient Encapsulation.

(On an unrelated note, it is clear that the classes are not named appropriately: the class names `SwingUtilities`, `SwingUtilities2`, and `SwingUtilities3` are quite general, and the difference between these classes is not evident from the class names.)

Example 4

Consider the following fields defined in `java.lang.System` class:

ANECDOTE

One of the authors was involved in reviewing an application for post-processing violations of static analyzers, clone detectors, and metric tools that was being developed at a start-up. The application generated summary reports that helped architects and project managers understand the code quality problems in their projects.

The application had three logical modules:

- `Parsing` module: Classes responsible for parsing the results from static analyzers, metric tools, and clone detectors.
- `Analytics` module: Classes responsible for analyzing the data such as aggregating them based on some criteria such as criticality of the violations.
- `Reporter` module: Classes responsible for generating reports and providing a summary to the architects and managers.

While inspecting the design and codebase, the author found an interesting class named `Data`. The main reason why this class existed is because the classes in `Reporting` module needed to know (for example, to create a summary report) the details resulting from the processing done by the `Parsing` and `Analytics` modules. To elaborate, the `Parsing` and `Analytics` modules provided direct and useful results that were used for summarizing the results in the `Reporting` module. The code in `Parsing` and `Analytics` module dumped the data structures into the `Data` class, and the code in `Reporting` module read this data from the `Data` class and created a summary as part of the final report that was generated. However, the way that this was realized was

ANECDOTE—Cont'd

by declaring all of the 23 fields of the `Data` class as public static (and non-final). The implication of such a design is that the details within `Analytics` module were unnecessarily available even to the `Parsing` module! This has an adverse impact on understandability and reliability. Such global data sharing provides an implicit form of coupling between the modules.

These members are declared final, but they can be "reset" using methods `setIn`, `setOut`, and `setErr`, respectively (these methods internally make use of native methods

```
public final static InputStream in = null;
public final static PrintStream out = null;
public final static PrintStream err = null;
```

to reset the final stream variables) provided in `java.lang.System` class. For instance, here is the declaration of the `setOut` method provided in `java.lang.System` class:

Since these fields are declared public, any code (i.e., other classes in JDK as well as application code) can access these fields directly. Hence, these fields are effec-

```
public static void setOut(PrintStream out)
```

tively global variables!

Common reason for making such fields public instead of providing getter methods is *performance*: since these fields are quite extensively used, it could considered as an unnecessary overhead to provide getter methods. However, providing such direct global access to fields is problematic. For illustration, let us consider one such problem.

The fields `out` and `err` are instances of type `PrintStream`. The `PrintStream` class was introduced in Java version 1.0 and supported only 8-bit ASCII values. Hence, to support Unicode, JDK 1.1.5 introduced `PrintWriter` class as a replacement to `PrintStream` class, which they intended to deprecate. However, fields of `PrintStream` type such as `System.out` and `System.err` are directly used by most of the Java applications to access `PrintStream`'s methods. For instance, every time we use `System.out.println`, we access the `println` method defined in `PrintStream` class! Hence, it is not possible to deprecate the entire `PrintStream` class.[1]

In summary, the deprecation of the `PrintStream` class was made difficult because of the Deficient Encapsulation smell in the `System` class which publicly exposed the data members `out` and `err` of type `PrintStream`.

[1] See Ref. [38] for an insightful discussion on the limited deprecation of the `PrintStream` class in JDK 1.1.5 and its later rollback in JDK 1.2.

CASE STUDY

Linux, a widely used open source operating system has attracted criticism from various quarters for its poor maintainability. One of the responsible factors for this is the large number of global variables in the source code. In a study, Yu et al. found that there are 99 global variables in version 2.4.20 of Linux [82]. Here, in particular, we are interested in a global variable named `current`.

This global variable was first introduced in version 1.0.0 of Linux and represents the currently running process. It is a pointer to a structure containing various fields that are used to describe the state of the process. Yu et al. [82] found that the global variable `current` is accessed (read and written) by 12 kernel modules and read by six other kernel modules. Furthermore, 1071 non-kernel modules have access to this global variable. The total number of read and write instances in kernel modules for this global variable is 382 and 114 respectively. Similarly, there are 6795 instances of read and 1403 instances of write for this global variable in non-kernel modules.

It is evident from the presented numbers that due to the global variable `current`, there is extremely strong coupling among various modules in Linux source code. This strong coupling makes it difficult to understand the functionality of the kernel and to debug the source code. This in turn makes changes to the source code error-prone.

The diagram (Figure 4.5) shows the strong coupling that exists due to the global variable `current` in Linux. It shows the coupling between kernel and non-kernel modules (files in C) due to the variable `current`. The arrows towards the box containing `current` are annotated. The annotation in the form of (x, y) indicates write and read instances for x and y, respectively. Similarly, annotation in the form of (x) indicates the number of read instances.

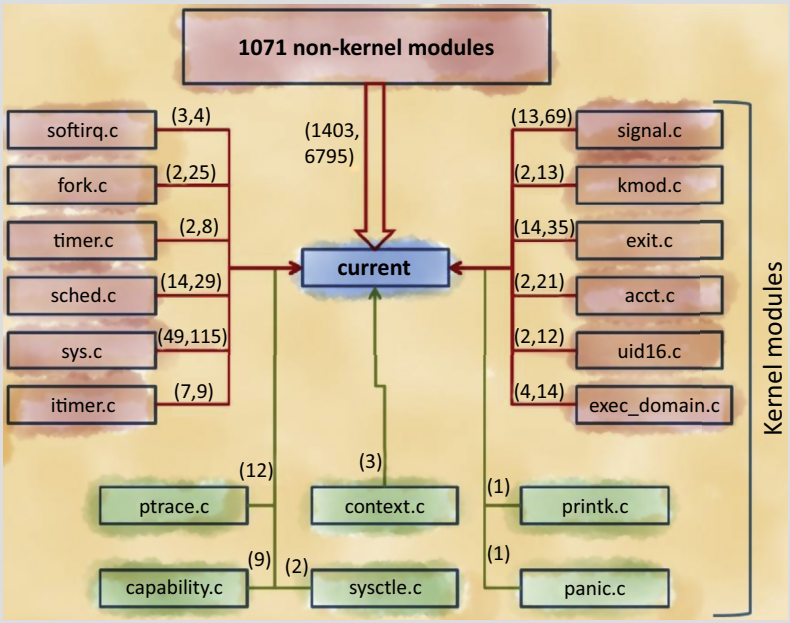

FIGURE 4.5

The use of global variable "current" in Linux kernel.

4.1.4 SUGGESTED REFACTORING

In the case of a public data member, "encapsulate field" refactoring can be applied to make the field private and provide accessor methods (i.e., get and set methods) as required. In case the methods that concern the implementation aspects of a class are made public, they must be made private (or protected if necessary).

Suggested refactoring for Examples 1 and 2

Make the data members in classes `Point` and `SizeRequirements` private and provide suitable accessor and mutator methods for the fields. For fields such as `alignment` in `SizeRequirements` class, provide the necessary validation for acceptable values in the mutator method.

Suggested refactoring for Example 3

Group the conceptually related methods in classes `SwingUtilities`, `SwingUtilities2`, and `SwingUtilities3` together in separate classes, and name the new classes based on the grouping, for example, `JavaFXUtilities`, `PaneUtilities`, `ComponentUtilities`, etc. Provide default access (i.e., package-private) access to these classes.

In the case of global variables, there are two variants possible:

- One or more members are made visible globally (by using `public static` in Java) even though only two or three classes access them
- One or more members are made visible globally (by using `public static` in Java) and a large number of classes access them

To refactor the first variant, passing the required variable as parameter could be considered. The second variant is more severe and trickier. One has to analyze existing abstractions and the responsibilities assigned to them to deal with this variant. It could be the case that the global variables have been introduced due to inappropriate abstractions and improper responsibility distribution. In such a case, the problem could be solved by identifying appropriate abstractions and encapsulating the variable within those abstractions. Now, clients can use these abstractions instead of globally visible data members.

Suggested refactoring for Example 4

To refactor this smell in `java.lang.System` class, one or more abstractions could be introduced to provide the functionality provided through the public members. For example, a class named, say, `Console`, could abstract the functionality that is provided by standard input, output, and error streams. This `Console` class could hide the implementation details (such as specific `Stream` classes used, kind of synchronization performed, etc.). Such a design would allow for changes in the implementation inside the `Console` abstraction, unlike the case in which data members (such as `out` and `err`) are directly exposed to the clients.

In fact Java 1.6 version introduced `java.io.Console` class that provides "methods to access the character-based console device." One can retrieve `Writer` or `Reader` objects associated with the `Console` object using `reader()` and `writer()` methods.

The class also provides methods such as `readLine()` and `printf()` for reading and writing strings to console, respectively. The readers and writers supported by this `Console` class wrap the actual input and output streams on which they operate. The `Console` class also uses locks internally to synchronize the reads and writes. In short, the `Console` class encapsulates the operations on input and output streams. This class is perhaps not meant to be a replacement of the original standard input, output, and error streams; however, it does illustrate what alternative design approach could be used instead of exposing streams directly to clients.

4.1.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—One of the main objectives of the principle of encapsulation is to shield clients from the complexity of an abstraction by hiding its internal details. However, when internal implementation details are exposed, the view of the abstraction may become complex and impact understandability. In the case when global data or data structures are used, it is very difficult to track and differentiate between read and write accesses. Furthermore, since the methods operating on the global data/data structures are not encapsulated along with the data, it is harder to understand the overall design.
- **Changeability** and **Extensibility**—Since the declared accessibility of members of the abstraction is more permissive, the clients of the abstraction may depend directly upon the implementation details of the abstraction. This direct dependency makes it difficult to change or extend the design. In the case of global variables, since encapsulation is lacking, any code can freely access or modify the data. This can result in unintended side effects. Furthermore, when defect fixes or feature enhancements are made, it can break the existing code. For these reasons, Deficient Encapsulation negatively impacts changeability and extensibility.
- **Reusability**—In the presence of Deficient Encapsulation smell, clients may directly depend on commonly accessible state. Hence, it is difficult to reuse the clients in a different context, since this will require the clients to be stripped of these dependencies.
- **Testability**—Globally accessible data and data structures are harder to test, since it is very difficult to determine how various code segments access or modify the common data. Furthermore, the abstractions that make use of the global data structures are also difficult to test because of the unpredictable state of the global data structures. These factors negatively impact testability of the design.
- **Reliability**—When an abstraction provides direct access to its data members, the responsibility of ensuring the integrity of the data and the overall abstraction is moved from the abstraction to each client of the abstraction. This increases the likelihood of the occurrence of runtime problems.

4.1.6 ALIASES

This smell is also known in literature as:

- "Hideable public attributes/methods" [55]—This smell occurs when you see public attributes/methods that are never used from another class but instead are used within the own class (ideally, these methods should have been private).
- "Unencapsulated class" [57]—A lot of global variables are being used by the class.
- "Class with unparameterized methods" [57]—Most of the methods in class have no parameters and utilize class or global variables for processing.

4.1.7 PRACTICAL CONSIDERATIONS

Lenient access in nested or anonymous classes

It is perhaps acceptable to declare data members or implementation-level methods as public in case of nested/anonymous classes where the members are accessible only to the enclosing class.

Performance considerations

Some designers make a conscious choice to make use of public data members citing efficiency reasons. Consider the class `Segment` from `java.swing.text` package (slightly edited to save space):

```
/** A segment of a character array representing a fragment of
text. It should be treated as immutable even though the array
is directly accessible. This gives fast access to fragments of
text without the overhead of copying around characters. This is
effectively an unprotected String. */
public class Segment implements Cloneable, CharacterIterator,
CharSequence {
   /** This is the array containing the text of interest. This
   array should never be modified; it is available only for
   efficiency. */
   public char[] array;
   /** This is the offset into the array that the desired text
   begins. */
   public int offset;
   /** This is the number of array elements that make up the text
   of interest. */
   public int count;
   // other members elided ...
}
```

This `Segment` class and its public fields are widely used within the `text` package. This design compromise would be acceptable assuming that the designers of the `Segment` class carefully weighed the tradeoff between performance benefits and the qualities that are violated because of this smell.

In general, designers should be wary of making such a decision. Often developers tend to declare fields public because they believe that direct access to those fields will be faster than through the accessor (i.e., get and set) methods. However, since current compilers and JIT compilers inline accessor methods, the performance overhead of using accessor methods is usually negligible.

4.2 LEAKY ENCAPSULATION

This smell arises when an abstraction "exposes" or "leaks" implementation details through its public interface. Since implementation details are exposed through the interface, it not only is harder to change the implementation but also allows clients to directly access the internals of the object (leading to potential state corruption).

It should be noted that although an abstraction may not suffer from Deficient Encapsulation (Section 4.1) (i.e., the declared accessibility of its members are as desired), it is still possible that the methods in the pubic interface of the abstraction may leak implementation details (see examples in Section 4.2.3).

4.2.1 RATIONALE

For effective encapsulation, it is important to separate the interface of an abstraction (i.e., "what" aspect of the abstraction) from its implementation (i.e., "how" aspect of the abstraction). Furthermore, for applying the principle of hiding, implementation aspects of an abstraction should be hidden from the clients of the abstraction.

When implementation details of an abstraction are exposed through the public interface (i.e., there is a violation of the enabling technique "hide implementation details"):

- Changes made to the implementation may impact the client code.
- Exposed implementation details may allow clients to get handles to internal data structures through the public interface, thus allowing clients to corrupt the internal state of the abstraction accidentally or intentionally.

Since implementation aspects are not hidden, this smell violates the principle of encapsulation; furthermore, the abstraction "leaks" implementation details through its public interface. Hence, we term the smell Leaky Encapsulation.

4.2.2 POTENTIAL CAUSES

Lack of awareness of what should be "hidden"

It requires considerable experience and expertise to discern and separate the implementation and interface aspects of an abstraction from each other. Inexperienced designers often inadvertently leak implementation details through the public interface.

Viscosity

It requires considerable thought and effort to create "leak-proof" interfaces. However, in practice, due to project pressures and deadlines, often designers or developers resort to quick and dirty hacks while designing interfaces, which leads to this smell.

Use of fine-grained interface

Often, this smell occurs because fine-grained methods are directly provided in the public interface of a class. These fine-grained methods typically expose unnecessary implementation details to the clients of the class. A better approach is to have logical coarse-grained methods in the public interface that internally make use of fine-grained methods that are private to the implementation.

4.2.3 EXAMPLES

Example 1

Consider a feature-rich e-mail application similar to Microsoft® Outlook® or Mozilla Thunderbird®. One of the features that such an application supports is maintaining a list of to-do items. Assume that the content of each to-do item is maintained in a `TodoItem` class. The class `TodoList` maintains the list of `TodoItems`. Figure 4.6 outlines the members in `TodoList` class (distilled to a very simple form to maintain focus on the smell under discussion).

One of the methods in `TodoList` class is the public method `getListEntries()`, which returns the list of to-do items maintained by that object. The problem is with the return type of the method i.e., `LinkedList`. The method exposes the internal detail that the `TodoList` class uses a linked list as the mechanism for maintaining the list of `TodoItems`.

Presumably, the application may have mostly inserts and deletes, and hence the class designer may have chosen a `LinkedList` for implementation in `TodoList` class. In the future, if it is found that the application performs search operations more frequently

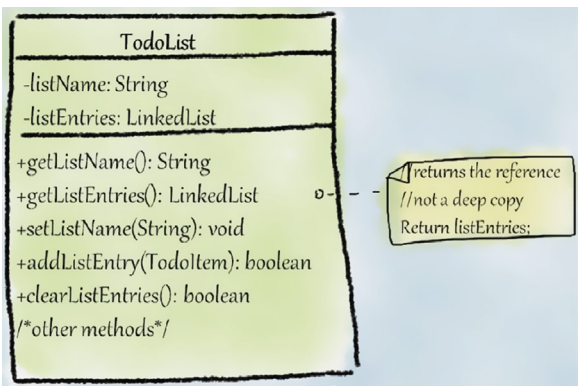

FIGURE 4.6

TodoList class exposing its internal use of linked list (Example 1).

than modifications to the list, it is better to change the implementation to use some other data structure such as an `ArrayList` or a `HashMap` (which provides faster lookup than a `LinkedList`). Furthermore, a new requirement tomorrow may require that at runtime the application should be able to switch to a specific implementation based on the context. However, since `getListEntries()` is a public method and returns `LinkedList`, changing the return type of this method may break the clients that are dependent on it. This suggests that it may be very difficult to support future modifications to the `TodoList` class.

Another serious problem with the `getListEntries()` method is that it returns a handle (i.e., reference) to the internal data structure. Using this handle, the clients can directly change the `listEntries` data structure bypassing the methods in `TodoList` such as `addListEntry()`.

To summarize, `TodoList` class "leaks" an implementation detail in its public interface, thereby binding the `TodoList` class to use a specific implementation.

ANECDOTE

A participant of the Smells Forum reported an example of this smell in an e-commerce application. In this application, the main logic involved processing of orders, which was abstracted in an `OrderList` class. The `OrderList` extended a custom tree data structure, since most use cases being tested by the team involved inserting and deleting orders.

When the application was first deployed, a customer filed a change request complaining that the application was not able to handle large number of orders and it "hanged." Profiling the application showed `OrderList` to be a "bottleneck class." The tree implementation internally used was not a self-balancing tree, and hence the tree often got skewed and looked more like a "list." To fix this problem, a developer quickly added a public method named `balance()` in the `OrderList` class that would balance the tree. He also ensured that the `balance()` method was invoked from the appropriate places in the application to address the skewing of the tree. When he tested the application, this solution appeared to solve the performance problem and so he closed the change request.

In a few months, more customers reported similar performance problems. Taking a relook at the class revealed that most of the time orders were being "looked-up" and not inserted or deleted. Hence the team now considered replacing the internal tree implementation to a hash-table implementation.

However, the `OrderList` class was tied to the "tree" implementation because it extended a custom `Tree` class. Furthermore, changing the implementation to use a more suitable data structure such as a hash table was difficult because the clients of the `OrderList` class used implementation-specific methods such as `balance()`! Although refactoring was successfully performed to use hash table instead of unbalanced tree in the `OrderList`, it involved making changes in many places within the application.

There are two key insights that are revealed when we reflect on this anecdote:

- Having the `OrderList` *extend* the custom tree instead of *using* the custom tree forced the `OrderList` to unnecessarily expose implementation details to the clients of the `OrderList` class. Due to the choice of using inheritance over delegation, the clients of the `OrderList` could not be shielded from the changes that arose when a different data structure, i.e., a hash table was used inside the `OrderList`.
- Adding `balance()` method in the public interface of the class is clearly a work-around. Such workarounds or quick-fix solutions appear to solve the problem, but they often backfire and cause more problems. Hence, one must invest a lot of thought in the design process.

Design smells such as Leaky Encapsulation can cause serious problems in applications, and we need to guard against such costly mistakes.

Example 2

Consider a class `CustomListBox`, which is a custom implementation of a ListBox UI control. One of the operations that you may perform on a `ListBox` object is to sort it. Figure 4.7 shows only the sort-related methods in `CustomListBox` class.

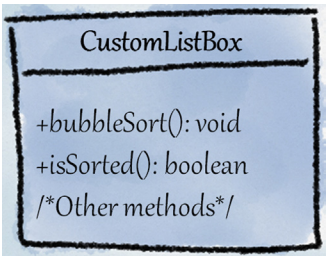

FIGURE 4.7

CustomListBox leaks the name of the sorting algorithm it uses (Example 2).

The problem with this class is that the name of the public method `bubbleSort()` exposes the implementation detail that the `CustomListBox` implementation uses "bubble sort" algorithm to sort the list. What if the sorting algorithm needs to be replaced with a different sorting algorithm, say, "quick sort" in the future? There are two possibilities:

- **Option 1**: Change the name of the algorithm to `quickSort()` and replace the algorithm accordingly. However, this change will break existing clients who are depending on the public interface of the class.
- **Option 2**: Replace only the implementation of the `bubbleSort()` method with the quick sort algorithm without changing the name of the existing method; i.e., the method name remains `bubbleSort()` but the method implements quick sort algorithm. The problem with this bad hack is that it will end up misleading the clients of this class that a "bubble sort" algorithm is being used because the name of the algorithm is mentioned in the method.

Thus both options are undesirable. In reflection, the root cause of the problem is that the name of the algorithm is "leaked" in the public interface of the class, thus leading to a Leaky Encapsulation smell.

4.2.4 SUGGESTED REFACTORING

The suggested refactoring for this smell is that the interface should be changed in such a way that the implementation aspects are not exposed via the interface. For example, if details of the internal algorithms are exposed through the public interface, then refactor the public interface in such a way that the algorithm details are not exposed via the interface.

It is also not proper to return a handle to internal data structures to the clients of a class, since the clients can directly change the internal state by making changes through the handle. There are a few ways to solve the problem:

- Perform deep copy and return that cloned object (changes to the cloned object do not affect the original object).
- Create an immutable object and return that cloned object (by definition, an immutable object cannot be changed).

If low-level fine-grained methods are provided in the public interface, consider making them private to the implementation. Instead, logical coarse-grained methods that make use of the fine-grained methods can be introduced in the public interface.

Suggested refactoring for Example 1

The suggested refactoring for the `ToDoList` example is to change the return type of the `getListEntries()` in such a way that it does not reveal the underlying implementation. Since `List` interface is the base type of classes such as `ArrayList` and `LinkedList`, one refactoring would be to use `List` as the return type of `getListEntries()` method. Or, a designer could choose to use `Collection` as the return type, which is quite general and does not reveal anything specific about the implementation.

However, it is also important to ensure that the clients of the `ToDoList` are not provided a handle to directly change its internal state. Therefore, consider returning a deep copy or a read-only copy of the internal list to the clients. At this point, let us take a step back and think about the clients of the `TodoList` class. Why would they be interested in `getListEntries()` method? If the end goal was to traverse the list and access each entry, then a better refactoring solution would be to return an `Iterator` instead. This kind of logical structured reasoning in practice will allow you to come up with a better and more effective design.

Suggested refactoring for Example 2

The suggested refactoring for the `CustomListBox` example is to replace the method name `bubbleSort()` with `sort()` (see Figure 4.8). With this, the `sort()` method can internally choose to implement any suitable sorting algorithm for sorting the

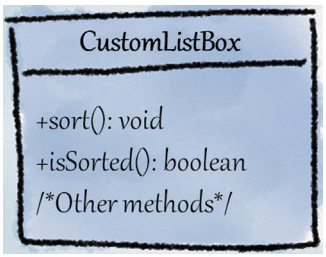

FIGURE 4.8

Suggested refactoring for CustomListBox (Example 2).

elements in the `CustomListBox`. For instance, the class `CustomListBox` can internally use Strategy pattern to employ the relevant algorithm at runtime.

Note that this class is part of the public API, so changing the method name will break the clients of the class. A work-around, to address this problem and the smell at the same time, is to introduce a new `sort()` method and deprecate the existing `bubbleSort()` method. Design is hard; API design is harder!

CASE STUDY

Consider the case of a software system in which a class implements operations on an image object. The operation to display an image completes in four steps that must be executed in a specific sequence, and the designer has created public methods corresponding to each of these steps: i.e., `load()`, `process()`, `validate()`, and `show()` in the `Image` class. Exposing these methods creates a constraint on using these four methods; specifically, the methods corresponding to these four steps—load, process, validate, and show—must be called in that specific sequence for the image object to be valid.

This constraint can create problems for the clients of the `Image` class. A novice developer may call `validate()` without calling `process()` on an image object. Each of the operations makes a certain assumption about the state of the image object, and if the methods are not called in the required sequence, then the operations will fail or leave the image object in an inconsistent state. Clearly, the `Image` class exposes "how" aspects of the abstraction as part of the public interface. It thus exhibits the Leaky Encapsulation smell.

How can this problem be addressed? One potential solution is to have a state flag to keep track of whether the operations are performed in a sequence. However, this is a really bad solution, since maintaining a state flag is cumbersome: if we maintain the state flag in the `Image` class, every operation must check for the proper transition and handle any exceptional condition.

At this juncture, we can take a step back and question why the four internal steps need to be exposed to the client who is concerned only with the display of the image. We can then see that the problem can be solved by exposing to the clients of the `Image` class only one method, say `display()`, which internally calls each of the four steps in sequence.

If we reflect upon this anecdote, we see that the underlying problem in this example is that the fine-grained steps were exposed as public methods. Hence, the key take-away here is that it is better to expose coarse-grained methods via the public interface rather than fine-grained methods that run the risk of exposing internal implementation details.

4.2.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—One of the main objectives of the principle of encapsulation is to shield clients from the complexity of an abstraction by hiding its internal details. However, when internal implementation details are exposed in the case of a Leaky Encapsulation smell, the public interface of the abstraction may become more complex, affecting its understandability.
- **Changeability and Extensibility**—When an abstraction "exposes" or "leaks" implementation details through its public interface, the clients of the abstraction may depend directly upon its implementation details. This direct dependency makes it difficult to change or extend the design without breaking the client code.

- **Reusability**—Clients of the abstraction with Leaky Encapsulation smell may directly depend upon the implementation details of the abstraction. Hence, it is difficult to reuse the clients in a different context, since it will require stripping the client of these dependencies (for reference, see Command Pattern [54]).
- **Reliability**—When an abstraction "leaks" internal data structures, the integrity of the abstraction may be compromised, leading to runtime problems.

4.2.6 ALIASES

This smell is also known in literature as:

- Leaking implementation details in API [49]—This smell occurs when an API leaks implementation details that may result in confusing users or inhibiting freedom to change implementation.

4.2.7 PRACTICAL CONSIDERATIONS

Low-level classes

Consider an example of embedded software that processes and plays audio in a mobile device. In such software, the data structures that store metadata about the audio stream (e.g., sampling rate, mono/stereo) need to be directly exposed to the middleware client. In such cases, when public interface is designed purposefully in this way, clients should be warned that the improper use of those public methods might result in violating the integrity of the object. Also, it is important to ensure that such classes are not part of the higher-level API, such as the one that is meant for use by the mobile application software, so that runtime problems can be avoided.

4.3 MISSING ENCAPSULATION

This smell occurs when implementation variations are not encapsulated within an abstraction or hierarchy.

This smell usually manifests in the following forms:

- A client is tightly coupled to the different variations of a service it needs. Thus, the client is impacted when a new variation must be supported or an existing variation must be changed.
- Whenever there is an attempt to support a new variation in a hierarchy, there is an unnecessary "explosion of classes," which increases the complexity of the design.

4.3.1 RATIONALE

The Open Closed Principle (OCP) states that a type should be open for extension and closed for modification. In other words, it should be possible to change a type's

behavior by extending it and not by modifying it (see case study below). When the variations in implementation in a type or a hierarchy are not encapsulated separately,

CASE STUDY ON OCP

Localization in Java is supported through the use of resource bundles (see `java.util.ResourceBundle` in Java 7). We can write code that is mostly independent of user's locale and provide locale-specific information in resource bundles.

The underlying process of searching, resolving, and loading resource bundles is complex. For most purposes, the default implementation provided by the `ResourceBundle` class is sufficient. However, for sophisticated applications, there may be a need to use a custom resource bundle loading process. For instance, when there are numerous heavyweight locale-specific objects, you may want to cache the objects to avoid reloading them. To support your own resource bundle formats, search strategy, or custom caching mechanism, `ResourceBundle` provides an extension point in the form of `ResourceBundle.Control` class. Here is the description of this class from JDK documentation:

"ResourceBundle.Control defines a set of callback methods that are invoked by the ResourceBundle.getBundle factory methods during the bundle loading process… Applications can specify ResourceBundle.Control instances returned by the getControl factory methods or created from a subclass of ResourceBundle.Control to customize the bundle loading process."

In summary, `ResourceBundle` supports a default process for locating and loading resource bundles that is sufficient for most needs. Furthermore, the class also provides support for customizing this bundle locating and loading process by providing a separate abstraction in the form of `ResourceBundle.Control` class. Thus, this design by default supports the standard policy and provides an extension point to plug-in a custom policy. Such a design follows OCP.

This example is also an illustration of what Alan Kay (Turing award winner and co-creator of Smalltalk) said about design: "Simple things should be simple, complex things should be possible."

it leads to the violation of OCP.

One of the enabling techniques for Encapsulation is "hide variations." This is similar to the "Variation Encapsulation Principle (VEP)" which is advocated by Gamma et al. [54]. Thus, when a type (or hierarchy) fails to encapsulate the variation in the implementation, it implies that the principle of encapsulation has been either poorly applied or not applied at all. Hence, we term this smell Missing Encapsulation.

4.3.2 POTENTIAL CAUSES

Lack of awareness of changing concerns

Inexperienced designers are often unaware of principles such as OCP and can end up creating designs that are not flexible enough to adapt to changing requirements. They may also lack the experience to observe or foresee the concerns that could change in the future (based on the planned product roadmap); as a result, these concerns may not be properly encapsulated in the design.

Lack of refactoring

As existing requirements change or new requirements emerge, the design needs to evolve holistically to accommodate the change. This is especially crucial when

designers observe or foresee major problems such as explosion of classes. The lack of refactoring in such cases can lead to this smell.

Mixing up concerns

When varying concerns that are largely orthogonal to each other are not separated but instead aggregated in a single hierarchy, it can result in an explosion of classes when the concerns vary.

Naive design decisions

When designers naively use simplistic approaches such as creating a class for every combination of variations, it can result in unnecessarily complex designs.

4.3.3 EXAMPLES

Example 1

Assume that you have an existing `Encryption` class that needs to encrypt data using an algorithm. There are various choices for such an algorithm including DES (Data Encryption Standard), AES (Advanced Encryption Standard), TDES (Triple Data Encryption Standard), etc. Figure 4.9 illustrates how the `Encryption` class encrypts data using the DES algorithm. Suppose a new requirement is introduced that requires data to be encrypted using the AES algorithm. A novice developer may come up with the design shown in Figure 4.10, in which he introduces different methods such as `encryptUsingDES()` and `encryptUsingAES()` in the `Encryption` class.

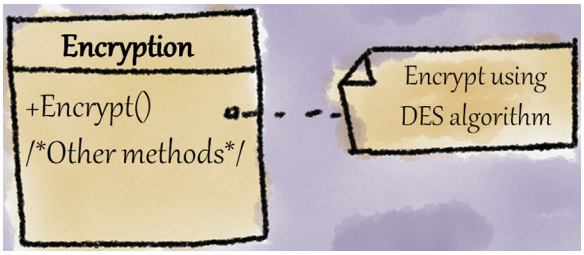

FIGURE 4.9

The class Encryption provides DES algorithm for encryption (Example 1).

There are many undesirable aspects of this design solution:

- The `Encryption` class becomes bigger and harder to maintain, since it implements multiple encryption algorithms even though only one algorithm is used at a time.
- It is difficult to add new algorithms and to vary the existing ones when a particular encryption algorithm is an integral part of the `Encryption` class.
- The algorithm implementations are tied to the `Encryption` class, so they cannot be reused in other contexts.

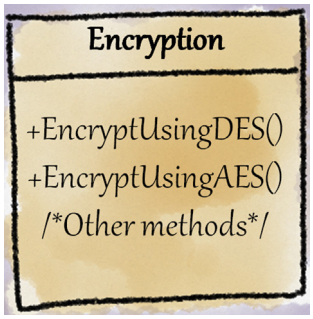

FIGURE 4.10

The class Encryption provides both DES and AES algorithms for encryption (Example 1).

In this example, encryption algorithms provide a service to the `Encryption` class. The `Encryption` class is tightly coupled to these services and is impacted when they vary.

Example 2

Consider a design that supports encrypting different kinds of content (such as images and text) as well as different kinds of algorithms (such as DES and AES). Figure 4.11 shows a commonly-seen simple design to model this solution.

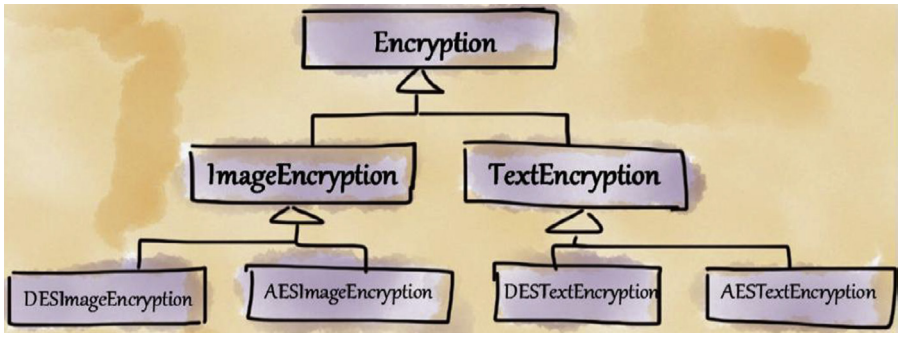

FIGURE 4.11

Hierarchy to support encryption of different kinds of content using different kinds of encryption algorithms (Example 2).

In this design, there are two variation points: the kind of content, and the kind of encryption algorithm that is supported. Every possible combination of these two variations is captured in a dedicated class. For instance, `DESImageEncryption` class deals with encrypting images using DES encryption algorithm. Similarly, `AESTextEncryption` class deals with encrypting text using AES encryption algorithm.

How does this design support a new type of content, a new encryption algorithm, or both? Figure 4.12 shows how the existing design will be extended to support TDES, a new encryption algorithm, and `Data`, a new type of content. Clearly, there

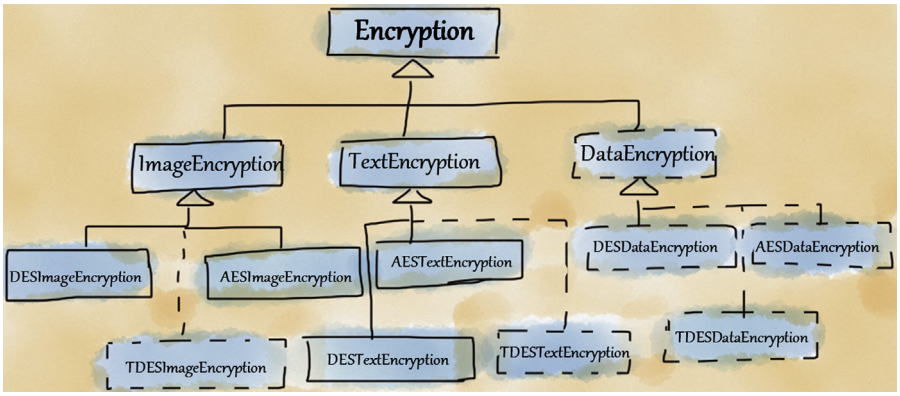

FIGURE 4.12

Class explosion in Encryption hierarchy when new kind of content and algorithm are added (Example 2).

is an explosion of classes. This is because the variations in the implementation are mixed together and have not been encapsulated separately, i.e., the design suffers from Missing Encapsulation smell.

4.3.4 SUGGESTED REFACTORING

The generic refactoring suggestion for this smell is to encapsulate the variations. This is often achieved in practice through the application of relevant design patterns such as Strategy and Bridge [54].

Suggested refactoring for Example 1

Let us consider what would happen if we were to use inheritance for refactoring Example 1. There are two options:

- **Option 1**: In this option, the `Encryption` class would inherit from `DESEncryptionAlgorithm` or `AESEncryptionAlgorithm` as needed and use the `encrypt()` method within. However, the problem with this solution is that the `Encryption` class would be tied to the specific encryption algorithm at compile time. A more serious problem is that the classes would not share an IS-A relationship, indicating a Broken Hierarchy smell (Section 6.8) (Figure 4.13).
- **Option 2**: Create subclasses named `DESEncryption` and `AESEncryption` that extend the `Encryption` class and contain implementation of DES and AES encryption algorithms, respectively. With this solution, the clients can hold reference to an `Encryption` class and point to the specific subclass object based on their need. By adding new subclasses, it is easier to add support for new encryption algorithms as well. However, the problem with this solution is that the `DESEncryption` and `AESEncryption` classes inherit other methods from the `Encryption` class. This reduces the reusability of the encryption algorithms in other contexts (Figure 4.14).

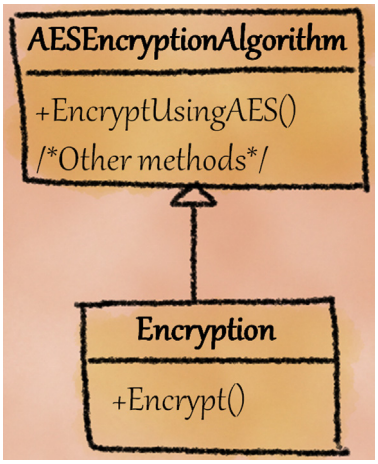

FIGURE 4.13

A possible refactoring for Example 1 (Option 1) which introduces Broken Hierarchy.

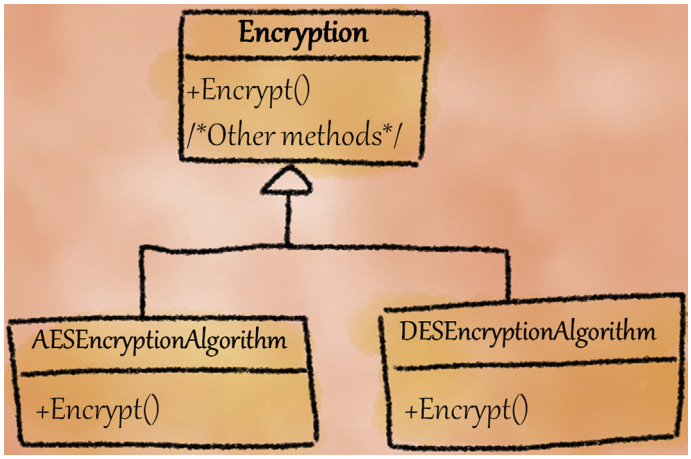

FIGURE 4.14

A possible refactoring for Example 1 (Option 2) using inheritance.

A better approach is to decouple the encryption algorithms from the `Encryption` class by employing the "factor out Strategy" refactoring [8]. The resulting design structure is shown in Figure 4.15.

In this design, an `EncryptionAlgorithm` interface is created. Classes named `DESEncryptionAlgorithm` and `AESEncryptionAlgorithm` implement the `EncryptionAlgorithm` interface and define the DES and AES algorithms, respectively. The `Encryption` class maintains a reference to the `EncryptionAlgorithm` interface.

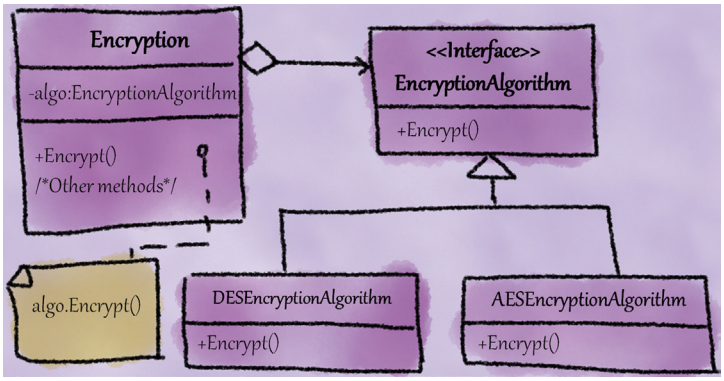

FIGURE 4.15

Suggested refactoring for Encryption class using Strategy pattern (Example 1).

This design structure offers the following benefits:

- An `Encryption` object can be configured with a specific encryption algorithm at runtime.
- The algorithms defined in the `EncryptionAlgorithm` hierarchy can be reused in other contexts.
- It is easy to add support for new encryption algorithms as and when required.

Suggested refactoring for Example 2

The main issue in the design (of Example 2) is that there are two orthogonal concerns that have been mixed up in the inheritance hierarchy. A refactoring suggestion is to use a structure similar to the Bridge pattern [54] and encapsulate the variations in these two concerns separately, as shown in Figure 4.16.

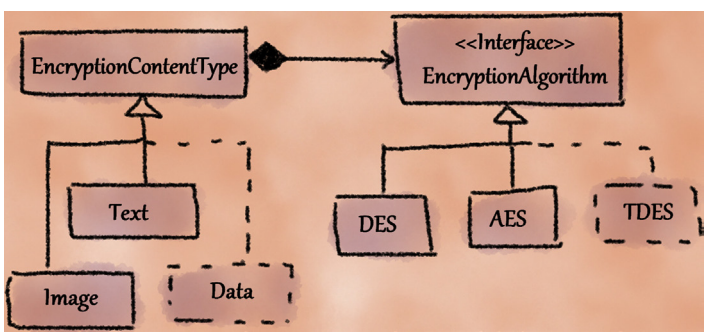

FIGURE 4.16

Suggested refactoring for supporting multiple kinds of content and algorithms (Example 2).

In this solution, the hierarchy rooted in `EncryptionContentType` class makes use of the algorithm implementations that are rooted in `EncryptionAlgorithm` interface. If a new content type named `Data` and a new encryption algorithm type TDES were to be introduced in this design, it would result in the addition of *only* two new classes (as shown in the Figure 4.16). In this way, the problem of explosion of the classes described in Example 2 can be addressed. Also, now, the encryption algorithm implementations rooted in `EncryptionAlgorithm` can be reused in other contexts.

4.3.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**—When there are numerous services (many of which may not be used) embedded in an abstraction, the abstraction becomes complex and difficult to understand. Furthermore, in cases where variation is not hidden in a hierarchy, the explosion of classes that results increases the complexity of design. These factors impact the understandability of the design.
- **Changeability** and **Extensibility**—When variations in implementation in a type or a hierarchy are not encapsulated separately, clients are tightly coupled to these different variations. This has two effects when a new or different variation must be supported. First, it is harder to change the clients when they need to use a different or new variation. Second, it may result in explosion of classes (see Example 2 above). If variations were to be encapsulated separately, the clients would be shielded from changes in the variations. In the absence of such encapsulation, changeability and extensibility are impacted.
- **Reusability**—When services are embedded within an abstraction, the services cannot be reused in other contexts. When two or more orthogonal concerns are mixed up in the hierarchy, the resulting abstractions that belong to the hierarchy are harder to reuse. For instance, in Example 2, when encryption content type and algorithm are mixed together, we no longer have specific algorithm classes that could be reused in other contexts. These factors impact the reusability of the type or hierarchy.

4.3.6 ALIASES

This smell is also known in literature as:

- "Nested generalization" [52]—This smell occurs when the first level in an inheritance hierarchy factors one generalization and the further levels "multiplies out" all possible combinations.
- "Class explosion" [22]—This smell occurs when number of classes explode due to extensive use of multiple generalizations.
- "Combinatorial explosion" [23]—This smell occurs when new classes need to be added to support each new family or variation.

4.3.7 PRACTICAL CONSIDERATIONS

None.

4.4 UNEXPLOITED ENCAPSULATION

This smell arises when client code uses explicit type checks (using chained if-else or switch statements that check for the type of the object) instead of exploiting the variation in types *already* encapsulated within a hierarchy.

4.4.1 RATIONALE

The presence of conditional statements (such as switch-case and if-else) in the client code that check a type explicitly and implement actions specific to each type is a well-known smell. Although this is a problem in general, consider the case when types are encapsulated in a hierarchy but are not leveraged. In other words, explicit type checks are being used instead of relying on dynamic polymorphism. This results in the following problems:

- Explicit type checks introduce tight coupling between the client and the concrete types, reducing the maintainability of the design. For instance, if a new type were to be introduced, the client will need to be updated with the new type check and associated actions.
- Clients need to perform explicit type check for all the relevant types in the hierarchy. If the client code misses checking one or more of the types in the hierarchy, it can lead to unexpected behavior at runtime. If runtime polymorphism were to be used instead, this problem could be avoided.

Such explicit type checks are an indication of the violation of the enabling technique "hide variations" and the principle of "encapsulate what varies" [54]. In the case of this smell, even though variation in types has been encapsulated within a hierarchy, it has not been exploited by the client code. Hence, we name this smell Unexploited Encapsulation.

4.4.2 POTENTIAL CAUSES

Procedural thinking in object-oriented language

Procedural languages such as C and Pascal do not support runtime polymorphism. Developers from procedural background think in terms of encoding types and use switch or chained if-else statements for choosing behavior. When the same approach is applied in an object-oriented language, it results in this smell.

Lack of application of object-oriented principles

Inexperienced software developers may be aware of object-oriented design principles, but they do not have a firm enough grasp of these concepts to actually put

them into practice. For instance, they may be familiar with the concept of hierarchy and polymorphism, but may not be aware of how to exploit these concepts properly to create a high-quality design. As a result, they may introduce type checks in their code.

4.4.3 EXAMPLE

In JDK, the hierarchy rooted in the abstract class `DataBuffer` is designed to wrap data arrays of various primitive types. `DataBuffer's` subclasses `DataBufferByte`, `DataBufferDouble`, `DataBufferFloat`, `DataBufferInt`, `DataBufferShort`, `DataBufferUShort` provide support for data arrays of specific primitive type (see Figure 4.17). The class `DataBuffer` also defines the constants `TYPE_BYTE`, `TYPE_DOUBLE`, `TYPE_FLOAT`, `TYPE_INT`, `TYPE_SHORT`, `TYPE_UNDEFINED`, and `TYPE_USHORT`. The JavaDoc comment for this class states, "the Java 2D API image classes use TYPE_BYTE, TYPE_USHORT, TYPE_INT, TYPE_SHORT, TYPE_FLOAT, and TYPE_DOUBLE DataBuffers to store image data."

What is interesting about encoding these data types is that these data types are used extensively in the `java.awt` package code base that supports images. Explicit type checking using a switch statement is also performed extensively for these data types (all the classes mentioned here are from `java.awt.image` package). For instance, `ComponentColorModel` class has 16 switch statements with case statements for some of these data types. Similarly, `Raster` class has 13 instances and `DirectColorModel` has 11 instances of switch-based explicit type checks. Classes `BandedSampleModel`, `ColorModel`, `ComponentSampleModel`, `IndexColorModel`, `MultiPixelPackedSampleModel`, `SampleModel`, and `SinglePixelPackedSampleModel` have at least one instance of switch-based explicit type check of these data types.

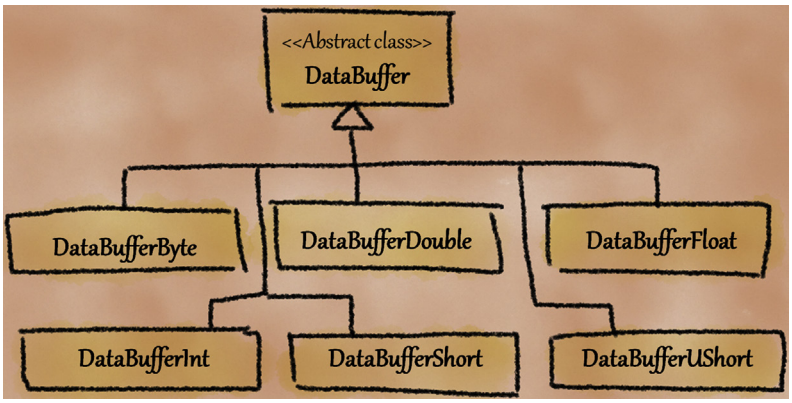

FIGURE 4.17

Explicit type checking is performed for the classes belonging to DataBuffer hierarchy.

Here is an example of how the explicit type check using a switch statement on concrete types is used in `ColorModel` class:

```
switch (transferType) {
  case DataBuffer.TYPE_BYTE:
    byte bdata[] = (byte[])inData;
    pixel = bdata[0] & 0xff;
    length = bdata.length;
    break;
  case DataBuffer.TYPE_USHORT:
    short sdata[] = (short[])inData;
    pixel = sdata[0] & 0xffff;
    length = sdata.length;
    break;
  case DataBuffer.TYPE_INT:
    int idata[] = (int[])inData;
    pixel = idata[0];
    length = idata.length;
    break;
  default:
    throw new
    UnsupportedOperationException("This method has not been "+
    "implemented for transferType" + transferType);
}
```

In this code, a data member `transferType` is defined as follows:

```
    /**
    * Data type of the array used to represent pixel values.
    */
    protected int transferType;
```

This extensive use of encoded type values instead of employing polymorphism to exploit the existing type hierarchy indicates an Unexploited Encapsulation smell.

4.4.4 SUGGESTED REFACTORING

A straight-forward refactoring for this smell would be to get rid of conditional statements that determine behavior and introduce polymorphic methods in the hierarchy.

Suggested refactoring for the Example

We describe here a potential refactoring for the `ColorModel` class that was discussed above. The refactoring here could be that instead of the `ColorModel`

class holding the `transferType` variable of type `int`, we could use a `DataBuffer` instance instead. Thus,

```
protected int transferType;
```

refactors to:

```
protected DataBuffer dataBuffer;
```

The appropriate type of `DataBuffer` could be provided via a constructor of the `ColorModel`. Methods such as `getPixel()` and `getSize()` could be defined in the types belonging to the `DataBuffer` hierarchy, and the whole switch along with associated cases can be reduced to the following two statements:

```
pixel = dataBuffer.getPixel();
length = dataBuffer.getSize();
```

Since the `dataBuffer` reference will point to the `DataBuffer` type that was passed through the constructor, the appropriate `getPixel()` and `getSize()` methods within that particular type will be invoked. The `ColorModel` class does not have to be aware of the actual type of `DataBuffer` anymore.

(Please note that the client code that instantiates `ColorModel` has to provide the specific type of `DataBuffer` in the `ColorModel` constructor. Therefore, this client code is responsible for checking the type of input data and creating the corresponding `DataBuffer` type. For this, a switch-case statement may be needed either in the client code or in a factory class [54]. However, unlike the original code in which switch-case statements were spread all over the codebase, only one such switch-case is needed in the entire codebase.)

This solution brings multiple benefits to the design. First, since `ColorModel` is not aware of the type of `DataBuffer` anymore, it is more loosely coupled to the `DataBuffer` hierarchy. Thus, it is not impacted if there is a change in any of the types or if a new type is added. Second, the refactored design is cleaner and leaner, since the unnecessary conditional statements and type checks have been removed.

4.4.5 IMPACTED QUALITY ATTRIBUTES

- **Changeability** and **Extensibility**—A hierarchy helps encapsulate variation, and hence it is easy to modify existing variations or add support for new variations within that hierarchy without affecting the client code. With type-based switches, the client code needs to be changed whenever an existing type is

changed or a new type is added. This adversely impacts changeability and extensibility.

- **Reusability**—Since the client code does type checks and executes certain functionality according to the type, reusability of the types in the hierarchy is impacted. This is because since some functionality is embedded in the client code, the types in the hierarchy do not contain all of the relevant functionality that may be needed to consider them as "reusable" units.
- **Testability**—When this smell is present, the same code segments may be replicated across the client code. This results in increased effort to test these fragments when compared to the effort if the code were to be provided within a hierarchy.
- **Reliability**—A common problem when using type-based checking is that programmers may overlook some of the checks. Such missing checks may manifest as defects.

ANECDOTE

The authors heard an interesting story at a software engineering conference from one of the attendees who worked for a telecommunications company. In one of his projects, he came across a class named `Pulse` which has an association with a hierarchy rooted in `Phone`. There are two subclasses of the `Phone` class namely `CellPhone` and `LandPhone`. While analyzing the design fragment, he realized that the code in the `Pulse` class was performing explicit type checks for types in the `Phone` hierarchy.

When he asked a project team member about this explicit type checking, the team member revealed that earlier they were making a polymorphic call to the `Phone` hierarchy to compute the pulse charges. However, a later requirement necessitated the computation of the data charges incurred by a cell phone. This required the `CellPhone` class to additionally support computation of data charges.

Now, in this project, the ownership of the `Phone` hierarchy was with the Device Team and the ownership of the `Pulse` class was with the Billing Team. These two teams were geographically distributed across two continents. To prevent sweeping changes to critical portions of the code, it was decided that changes to the critical portions could be made only by the teams responsible for them. Thus, any changes to the `Phone` hierarchy (and this included the change to `CellPhone` to support computation of data charges) could be made only by the Device Team. Upon receiving the requirement, the Billing Team (which was responsible for the `Pulse` class) requested the Device Team to make changes to the `CellPhone` class. Since the Device Team already had a huge backlog of change requests, they could not fulfill this requirement in time for the product release. Left with little choice and under time pressure, the Billing Team implemented the requirement by explicitly checking for the type of `Phone` and calling existing methods of `CellPhone` to compute data charges in the `Pulse` class itself! This implementation led to the Unexploited Encapsulation smell.

The key learning from this anecdote is how the dynamics of two teams working on the same product may introduce smells. A proper coordination between the Device and Billing teams could have avoided this problem. Furthermore, it is understandable that the inefficient solution adopted by the Billing Team was necessary, given the time pressure, but adequate measures should have been taken to document it and ensure that this problem was rectified later. A follow-up should have been done after the product release to ensure that the `CellPhone` and `Pulse` classes were modified appropriately and the Unexploited Encapsulation smell was removed. However, over and over again, project teams forget about the shortcuts that they have used and fail to rectify their mistakes. This negligence builds up technical debt over time and eventually results in steep increase in maintenance costs for the project.

4.4.6 ALIASES

This smell is also known in literature as:

- "Simulated polymorphism" [28]—This smell occurs when the code does conditional checks and chooses different behavior based on the type of the object to simulate polymorphism.
- "Improper use of switch statement" [7]—This smell occurs when switch statements are used instead of employing runtime polymorphism.
- "instanceof checks" [26]—This smell occurs when code has concentration of instanceof operators in a code block.

4.4.7 PRACTICAL CONSIDERATIONS

None.