CHAPTER

Modularization Smells

5

The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition.

To understand what modularization means and why it is important, let us consider the very case of this book! Let us imagine that we rewrite this book and explain it like one long story without any breaks. The resulting book will not have chapters, sections, or subsections. Each smell will be described in an ad hoc manner and there will be no definite order to the discussion of smells in the book. Further, there will be no cross references between smells (Figure 5.1).

How would readers react to such a book? Would this be an easy read? Clearly, readers would find it difficult to distinguish between smells. They will have to flip pages back and forth to gain a comprehensive understanding of smells. Further, they will be unable to gain a proper understanding of the various design principles

FIGURE 5.1

A book with no chapters and sub-sections is difficult to read.

Refactoring for Software Design Smells. http://dx.doi.org/10.1016/B978-0-12-801397-7.00005-9

because the principles would be intermingled and the description of those principles spread throughout the book. In summary, these problems would give readers a harrowing experience and reduce the usefulness of the book.

However, the current book organizes and aggregates smells based on the principles that are violated. The book "decomposes" the subject material into three logical parts, namely, the introduction and background, the catalog of smells, and the reflections. Additionally, each part is divided into several chapters that deal with a specific topic. This allows readers to quickly navigate to the particular design principle and the smells that are "categorized" under that principle. Additionally, most discussions about a design principle are "localized" in the introductory section of the concerned chapter, which allows readers to access all the information about a particular design principle at a single place in the book. In other words, the current book demonstrates the application of techniques such as decomposition and localization to achieve good modularization.

Similarly, it is important to follow the principle of modularization during software design. It should be pointed out that while modularization is generally considered to be a system-level concern (and concerns how we package abstractions to create logical modules), we use the term *module* here to mean *abstractions at the class level*, specifically, concrete classes, abstract classes, and interfaces. Thus, the goal behind modularization in the context of this book is to create cohesive and loosely coupled abstractions.

Figure 5.2 shows the enabling techniques that help apply the principle of modularization. These are:

- **Localize related data and methods.** Each abstraction should be cohesive in nature, i.e., the abstraction should keep related data and associated methods together.
- **Decompose abstractions to manageable size.** Break large abstractions into smaller ones that are moderate in size (i.e., neither too small, nor too large in size). For instance, a huge class not only makes it difficult for the reader to understand it, but also makes changes difficult because of possible interweaved responsibilities implemented by the class.
- **Create acyclic dependencies.** Abstractions should not be cyclically-dependent on each other. In other words, if a dependency graph is created, it should be free

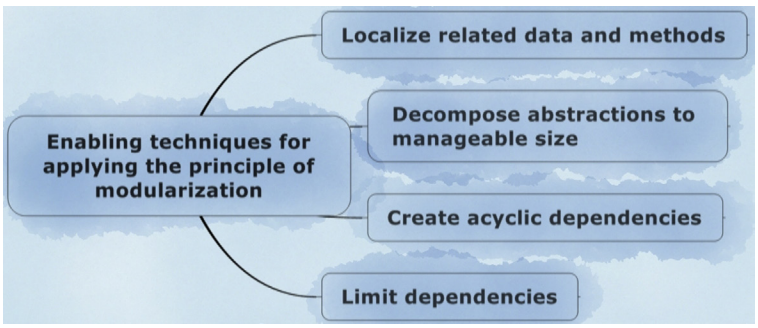

FIGURE 5.2

Enabling techniques for the principle of modularization.

of cycles. Otherwise, a change in an abstraction may result in a ripple effect across the entire design.

- **Limit dependencies.** Create abstractions with low fan-in and fan-out. *Fan-in* refers to the number of abstractions that are dependent on a given abstraction. Thus, a change in an abstraction with high fan-in may result in changes to a large number of its clients. *Fan-out* refers to the number of abstractions on which a given abstraction depends. A large fan-out indicates that any change in any of these abstractions may impact the given abstraction. Thus, to prevent a ripple effect due to potential changes, it is important to reduce the number of dependencies between abstractions in the design.

Each smell described in this chapter maps to a violation of an enabling technique. Figure 5.3 provides an overview of the smells that violate the principle of modularization and Table 5.1 provides an overview of mapping between the design smells and the

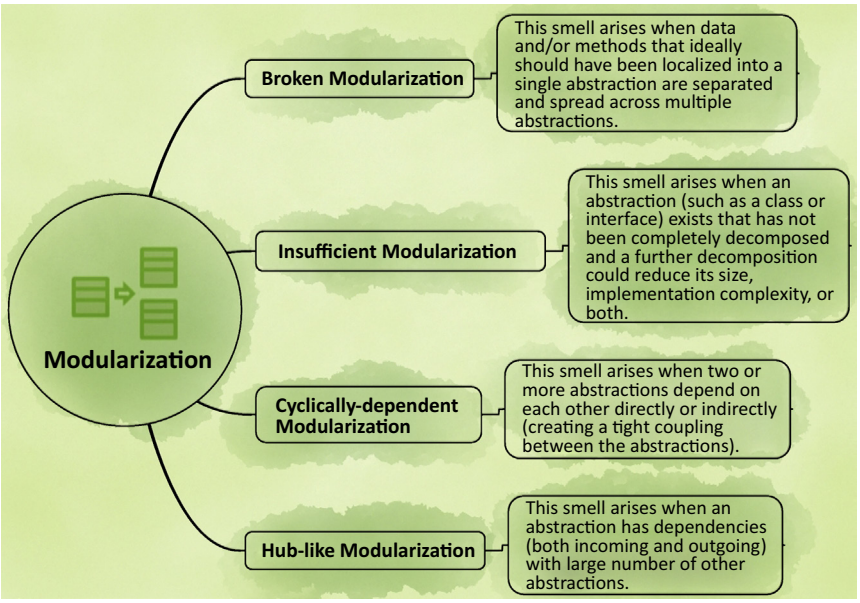

FIGURE 5.3

Smells resulting from the violation of the principle of modularization.

Table 5.1 Design Smells and Corresponding Violated Enabling Technique

Design Smells	Violated Enabling Technique
Broken modularization (5.1)	*Localize related data and methods*
Insufficient modularization (5.2)	*Decompose abstractions to manageable size*
Cyclically-dependent modularization (5.3)	*Create acyclic dependencies*
Hub-like modularization (5.4)	*Limit dependencies*

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.

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

5.1 BROKEN MODULARIZATION

This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.

This smell commonly manifests as:

- Classes that are used as a holder for data but have no methods operating on the data within that class.
- Methods in a class that are more interested in members of other classes.

5.1.1 RATIONALE

One of the key enabling techniques for modularization is to "localize related data and methods." Lanza and Marinescu [21] recommend that "data and operations should collaborate harmoniously within the class to which they semantically belong." An abstraction suffering from this smell violates this enabling technique when the abstraction has only data members and the methods operating on the data are provided in some other abstraction(s).

When members that should ideally belong in a single abstraction are spread across multiple abstractions, the result is a tight coupling among the abstractions. Hence, this smell violates the principle of modularization. Since related members are "broken" and are made part of different abstractions, this smell is named Broken Modularization.

5.1.2 POTENTIAL CAUSES

Procedural thinking in object-oriented languages

Procedural languages provide language features such as "structs" (in C) and "record" (in Pascal) that hold data members together. Functions process the common data stored in structs/records. When developers from procedural language backgrounds such as C or Pascal move to an object-oriented language, they tend to separate data from functions operating on it, thereby resulting in this smell.

Lack of knowledge of existing design

Large real-world projects have a complex design and have a codebase that is spread over numerous packages. In such projects, developers usually work on a small part of the system and may not be aware of the other parts of the design. Due to this, while implementing a new feature, developers may be unaware of the most appropriate classes where data/methods should be placed; this may lead to members being misplaced in the wrong classes.

5.1.3 EXAMPLES

Example 1

Consider an application that manages peripheral devices remotely over a network. In this application, data related to a device is stored in a class named `DeviceData`. The methods for processing the device data are provided in a class named `Device`. What is interesting about these two classes is that `DeviceData` only has public data members with no methods, and device class holds an object of type `DeviceData` and provides methods for accessing and manipulating that data member (Figure 5.4). Clearly, since the data and methods that ideally should have been localized in a single class have been separated across `Device` and `DeviceData` classes, this design fragment is an instance of a Broken Modularization smell.

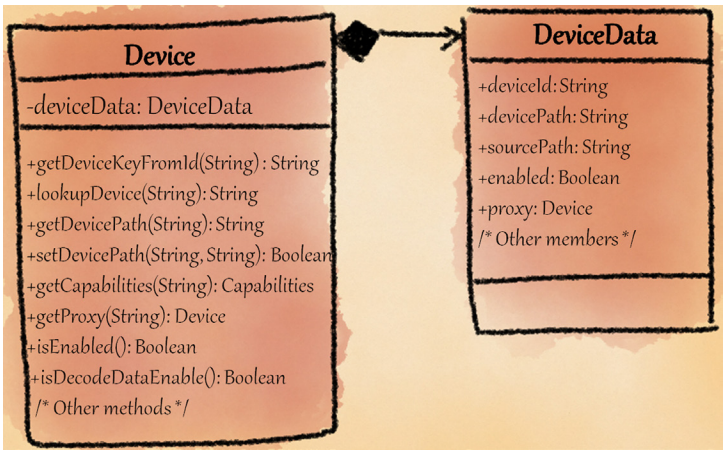

FIGURE 5.4

Class diagram in a device management application (Example 1).

Example 2

In an application for customer order processing, a key class was named as `Order`. Figure 5.5 shows the simplified version of this `Order` class; other classes interacting with the `Order` class such as `Customer`, `Payment`, `ShoppingCart`, `LineItem`, and `Product` are not shown in this figure. What was interesting about this design was that the `Order` class had only data members and the code operating on the data such as code that validates the order, calculates tax, and changes the order status was dispersed throughout the application. Clearly, the `Order` class along with its associated classes exhibits the Broken Modularization smell.

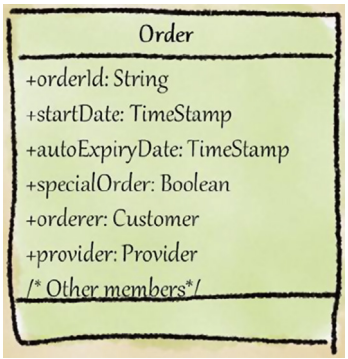

FIGURE 5.5

Order class that contains only public data members (Example 2).

As a result of this smell, maintaining the code related to `Orders` was a nightmare—for a fix or an enhancement, the code needed to be changed in multiple places. Further, since the members were public, any client code could change an `Order` object and trigger illegal state transitions. For instance, if a valid order state transition were to be "new" -> "paymentreceived" -> "processing" -> "shipped" -> "delivered" -> "closed", a client code could directly modify the `Order` object to move its state from "new" state to "closed" state.

ANECDOTE

During a lecture on software architecture and design that one of the authors was delivering one of the participants shared an experience from his project. This participant was associated with the development of a multiversion configuration tool for devices. This tool included a Graphical User Interface (GUI) editor, which first allowed the user to input configuration parameters and then invoked "checks" to verify whether the input data were acceptable. The software architecture was typical—there was a User Interface (UI) layer, a Business Logic layer, a Data layer, and a cross-cutting Utility layer. In version 1.0 of the editor, the "checks" were coded in the UI layer. In version 1.2, additional "checks" were added in the Utility layer. The new version 2.0 finally had "check" functionality in the UI, Business Logic, and Utility layer.

Unfortunately, there was no refactoring planned or executed to ensure quality of design as the software evolved. The lack of refactoring during the subsequent versions resulted in a design where the check functionality was dispersed across multiple components across layers. Every time a change or fix was needed in the check functionality, one would need to inspect all three layers where the functionality was implemented. Needless to say, it became extremely complex to manage and extend the check functionality. Eventually, due to lack of *localization* of these checks in a single layer, maintaining the check functionality resulted in massive effort and delays.

Even though this anecdote talks about an architectural problem, it is very relevant to this book:

- It highlights how even small design decisions can have a huge impact on the architecture and the project.
- It underlines the importance of adhering to fundamental principles and techniques such as localization, separation of concerns, and high cohesion and low coupling.

5.1.4 SUGGESTED REFACTORING

When the design is procedural in style, and has numerous data classes, apply "convert procedural design to objects" refactoring [7]. In this refactoring, we move the data members and the behavior associated with the data members into the same class.

- If a method is used more by another class (Target class) than by the class in which it is defined (Source class), apply "move method" refactoring and move that method from Source class to the Target class (see Figure 5.6).
- If a field is used more by another class (Target class) than the class in which it is defined (Source class), apply "move field" refactoring and move that field from Source class to the Target class (see Figure 5.6).

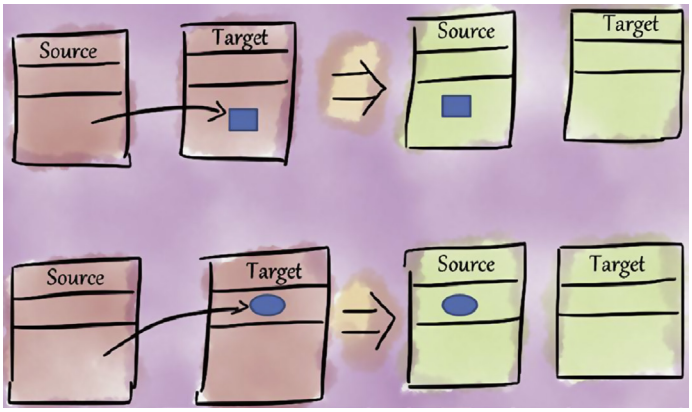

FIGURE 5.6

Suggested refactoring for Broken Modularization smell.

Suggested refactoring for Example 1

Since data and behavior for the same responsibility are split across `DeviceData` and `Device`, a natural refactoring is to combine them together in a single class (see Figure 5.7).

Suggested refactoring for Example 2

The following set of refactorings can be applied to refactor the example:

- `Order` class has multiple responsibilities. So, it needs to be broken into multiple classes such as `OrderItem`, `OrderDetail`, and `Payment`.
- Data members in `Order` need to be made private.
- Methods operating on the `Order` members need to be provided within the `Order` class itself. For example, `validateOrder()` could be a method in this class.
- Members not directly relating to `Order` should be moved to appropriate classes.

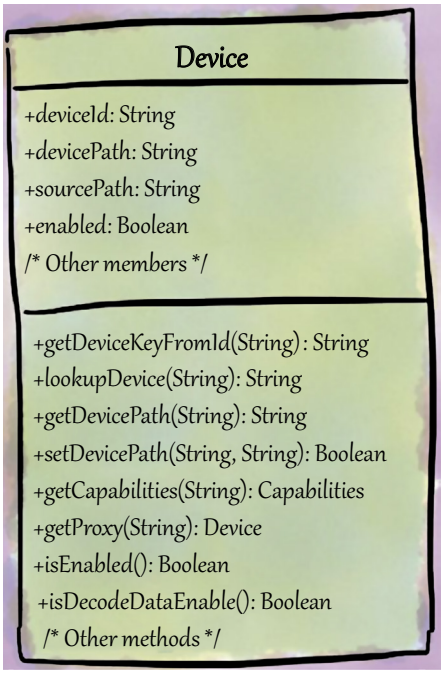

FIGURE 5.7

Suggested refactoring in device management application (Example 1).

It should be noted that grouping all seemingly related methods into a single abstraction to address Broken Modularization can lead to a Multifaceted Abstraction (Section 3.4) or Insufficient Modularization (Section 5.2) smell. Hence, due caution needs to be exercised while refactoring this smell.

5.1.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability**: In case of Broken Modularization smell, to understand how a particular functionality is realized, one needs to understand all the associated methods and data that are spread across abstractions. This impacts understandability of the design.
- **Changeability** and **Extensibility:** When members that should ideally belong to an abstraction are separated across abstractions, supporting changes or enhancements becomes difficult because modifications may need to be made across multiple abstractions. This impacts changeability and extensibility of the design.
- **Reusability** and **Testability:** In the case of this smell, when we want to reuse or test a particular functionality, we have to use multiple abstractions together instead of a single abstraction. This impacts reusability and testability of the design.

- **Reliability:** When data and methods operating on that data are split across abstractions, encapsulation of those abstractions may be compromised and implementation details could get exposed. Further, when methods are misplaced in other abstractions, those methods are exposed to the implementation details (such as data structures and algorithms) of the abstractions in which they are placed. These factors could lead to defects affecting reliability.

5.1.6 ALIASES

This smell is also known in literature as:

- Class passively stores data [59]: This smell occurs when a class passively stores data and does not provide methods to operate on the data.
- Data class [7, 10, 57, 76]: This smell occurs when a class is used as a "dumb" data holder without complex functionality (but the class is usually heavily relied upon by other classes in the system).
- Data records [77]: This smell occurs when classes contain only data members without any methods.
- Record (class) [78]: This smell occurs when a class looks and feels much like Pascal record types; in this case, the class has all of its fields public and has no methods other than constructors and methods inherited from `Object` class.
- Data container [25]: This smell occurs when one class holds all the necessary data (and is called the Data Container), whereas the second class interacts with other classes implementing functionality related to the data of the first class.
- Misplaced operations [8]: This smell occurs when unexploited cohesive operations are outside (i.e., in other classes) instead of inside the same class.
- Feature envy [7, 76]: This smell occurs when there is a method that seems more interested in a class other than the one it is actually in.
- Misplaced control [30]: This smell occurs when a piece of functionality is unjustifiably separated from the data on which it operates.

5.1.7 PRACTICAL CONSIDERATIONS

Auto-generated code

The code generated from auto-code generators (from higher level models) often consists of a number of data classes. In the context of auto-generated code, it is not a recommended practice to directly change the generated code, since the models will be out-of-sync with the code. Hence, it may be acceptable to live with such data classes in generated code.

Data Transfer Objects (DTOs)

Often, a Data Transfer Object (DTO) is used to transfer data between processes to reduce the number of calls that would have been otherwise required in the context of remote interfaces. DTOs aggregate data and lack behavior; this is done consciously to facilitate serialization [84].

5.2 INSUFFICIENT MODULARIZATION

This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.

There are two forms of this smell:

- **Bloated interface:** An abstraction has a large number of members in its public interface.
- **Bloated implementation:** An abstraction has a large number of methods in its implementation or has one or more methods with excessive implementation complexity.

The smell could also appear as a combination of these two forms, i.e., "bloated interface and implementation."

5.2.1 RATIONALE

Modularization concerns the logical partitioning of a software design so that the design becomes easy to understand and maintain. One of the key enabling techniques for effective modularization is to "decompose abstractions to manageable size." In this context, Lanza and Marinescu [21] recommend that "operations and classes should have a harmonious size, i.e., they should avoid both size extremities." When an abstraction has complex methods or a large number of methods, it violates this enabling technique and the principle of modularization. Since the abstraction is inadequately decomposed, we name this smell Insufficient Modularization.

MULTIFACETED ABSTRACTION VERSUS INSUFFICIENT MODULARIZATION

Note that Multifaceted Abstraction smell (Section 3.4) is different from Insufficient Modularization smell. In Multifaceted Abstraction, an abstraction addresses more than one responsibility, thereby violating the Single Responsibility Principle (SRP). In Insufficient Modularization, an abstraction violates the Principle of Decomposition, indicating that the abstraction is large, complex, or both. Interestingly, these two smells often occur together: It is common to see a large and complex abstraction that has multiple responsibilities. However, they are different smells: An abstraction could have a single responsibility, but could still be large and complex. Similarly, an abstraction could be small, but could still have multiple responsibilities.

5.2.2 POTENTIAL CAUSES

Providing centralized control

A typical cause of bloated implementation is centralizing control and assigning a large amount of work to a single abstraction or a method in an abstraction.

Creating large classes for use by multiple clients

Often, software developers create a large abstraction so that it can be used by multiple clients. However, creating a single large abstraction that fits the needs of multiple clients leads to many problems, such as reduced changeability and extensibility of the design (see the Impacted Quality Attributes section).

Grouping all related functionality together

Often, inexperienced developers tend to group together and provide all related functionality in a single class or interface without understanding how the Single Responsibility Principle (SRP) should be properly applied. This results in bloated interfaces or classes.

5.2.3 EXAMPLES

Example 1: bloated implementation

An example of Insufficient Modularization with bloated implementation is `java.net.SocketPermission` in JDK 1.7. The problem with the `SocketPermission` class is that the sum of Cyclomatic complexities of its methods[1] is 193! In particular, one of its private methods `getMask()` has a Cyclomatic complexity of 81 due to its complex conditional checks with nested loops, conditionals, etc.

Example 2: bloated interface and implementation

The `java.awt.Component` class is an abstraction for graphical objects such as buttons and checkboxes. This abstract class is an example of Insufficient Modularization that has bloated interface as well as bloated implementation. It is a massive class with 332 methods (of which 259 are public), 11 nested/inner classes, and 107 fields (including constants) (see Figure 5.8). The source file of this class spans 10,102 lines of code! The sum of Cyclomatic complexity of its methods is 1143, indicating that the implementation is dense. Many methods belonging to the class have high Cyclomatic complexity; for example, the method `dispatchEventImpl(Event)` has a Cyclomatic complexity of 65.

Other examples from JDK 1.7 that exhibit bloated interface and implementation include:

- `java.swing.JTable` (Figure 5.9) has 44 attributes and 203 methods, and 9608 lines of code (as of JDK version 1.7).
- `java.lang.BigInteger` has 28 fields and 103 methods.
- `java.util.Calendar` has 81 fields and 71 methods.

[1] The sum of Cyclomatic complexity of methods in a class is also known by the metric name Weighted Method per Class (WMC).

ANECDOTE

One of the participants at the Smells Forum shared her experience that highlighted how the viscosity of the environment can lead to the introduction of Insufficient Modularization smell. She had just joined a globally distributed software development project concerning a critical clinical workflow product. The ownership of the code base was with the central team located in Country A, and she was a part of the offshore development team located in Country B.

In the first few weeks of her involvement, she realized that there were a number of smells in the design and code. For instance, she came across a class in the source code that was very long and dense – it had around 40,000 source lines of code and the Weighted Methods per Class - i.e., the sum of Cyclomatic complexities of the methods of a class – exceeded 2000! In other words, this class was a clear example of Insufficient Modularization. When she approached her team members, she realized that they were already aware of the size and complexity of this class. In fact, she also came to know that this class was prone to defects and was subjected to frequent changes.

Being new to the project, she was puzzled about why this class was so huge and why no refactoring had been performed on it so far. Upon further probing, she found out that the real problem was due to the process being followed in the project! There were two aspects that contributed to this problem. First, to prevent unwarranted modifications to the critical product, the project relied on a stringent process to control changes to the source code. As per this process, whenever a new class was introduced by the team in Country B, it needed to be approved by the central team in Country A because it owned the code base. This approval process was a long and arduous affair requiring multiple email and telephone interactions to champion the need for adding a new class.

Second, the project management was understandably very concerned about the timely release of this product in the market and continuously pressurized the team in Country B to finish coding and bug-fixing activities as early as possible. Naturally, in such a situation, the team in Country B wanted to avoid the time-consuming approval process for new classes. Consequently, the team decided not to introduce any new classes and instead decided to merge all new code in existing classes. This led to the bloating of existing classes in the design.

The nature of the change approval process also discouraged refactoring. For instance, refactoring the class with 40000 lines would require breaking it up into several smaller cohesive classes. Since the approval for all these newly-created classes would require a long time, the team in Country B was not keen to refactor such large and complex classes. Since refactoring was not taken up at regular intervals, even during the maintenance phase, the existing classes continued to bloat and became increasingly bug-prone.

There are two key take-aways from this anecdote:

- Although software processes are meant to facilitate the creation of higher quality software, if they are difficult to follow or not user-friendly, people will either bypass the process, or avoid it completely, or take short-cuts thus affecting the quality of software. Organizations should, therefore, strive to make software processes user-friendly.
- Project management is often unaware of the impact of software processes. In the above case, if the project management were to be aware of the nature of the change approval process and its impact on the schedule, they could have taken appropriate steps to avoid incurring technical debt. For instance, they could have optimized the change approval process (perhaps by having it done in the same country) so that the schedule is met. Alternately, they could have modified the schedule to ensure that the team in Country B feels encouraged to follow good design practices and the change approval process set in place.

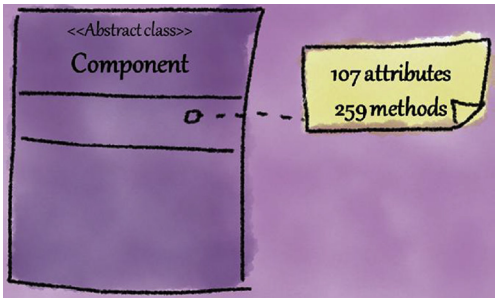

FIGURE 5.8

java.awt.Component class is large and dense (Example 2).

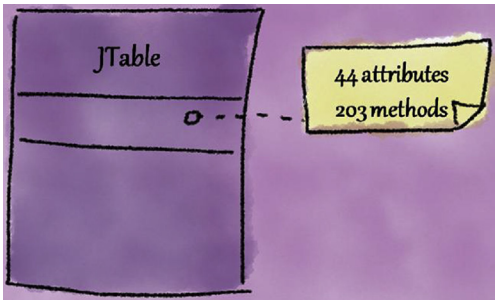

FIGURE 5.9

java.swing.JTable class is large and dense (Example 2).

5.2.4 SUGGESTED REFACTORING

Refactoring for bloated interface

- If a subset of the public interface consists of closely related (cohesive) members, then extract that subset to a separate abstraction.
- If a class consists of two or more subsets of members and each set of members is cohesive, split these subsets into separate classes.
- If the class interface serves multiple clients via client-specific methods, consider applying the Interface Segregation Principle (ISP) to separate the original interface into multiple client-specific interfaces.

Refactoring for bloated implementation

- If the method logic is complex, introduce private helper methods that help simplify the code in that method.
- In case an abstraction has both Multifaceted Abstraction smell as well as Insufficient Modularization smell, encapsulate each responsibility within separate (new or existing) abstractions.

Suggested refactoring for Example 1

Create helper methods for the complex methods such as `getMask()` in `java.net.SocketPermission`. For example, the private `getMask()` method has code for skipping whitespace in the passed string argument, which could be extracted to a new helper method. Interestingly, `getMask()` method also has duplicate blocks of code for checking characters in four known strings: `CONNECT`, `RESOLVE`, `LISTEN`, and `ACCEPT`. One possible refactoring is to create a helper method named `matchForKnownStrings()` that checks for the characters in these four strings. Now, the code within the `getMask()` method can be simplified by making four calls to `matchForKnownStrings()` instead of duplicating code four times for making these checks.

Suggested refactoring for Example 2

The abstract class `Component` provides default functionality to create lightweight components that can be defined by extending the `Component` class. It assembles all the required default functionality that a new component may reuse. It covers a number of concerns such as event listening, component layout and positioning, focus, fonts, and graphics configuration, which makes the class huge.

Hence, a suggested refactoring is to separate out these concerns into different abstractions and make the `Component` class use these abstractions via delegation. For instance, event listening can be separated out as an interface (say, `ComponentEvents`) and a default implementation (say, `DefaultComponentEvents` class) of the event listening functionality for components can be provided by realizing this interface. The `Component` class can now use `DefaultComponentEvents` class via delegation for listening to events. Any custom support for event listening can now be realized by providing a suitable implementation of the `ComponentEvents` interface.

Additionally, methods such as `getFontMetrics()`, `createImage()`, `createVolatileImage()` can be moved to `FontMetrics`, `Image`, and `VolatileImage` classes, respectively. To summarize, by suitably extracting abstractions or moving methods to classes, the Insufficient Modularization smell in `Component` class can be addressed.

5.2.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability:** Clients find bloated interfaces hard to comprehend and use. Similarly, it is difficult for developers to understand and maintain an abstraction with bloated implementation. The complexity of the abstraction thus impacts its understandability.
- **Changeability** and **Extensibility:** When an abstraction has a bloated interface, typically there are numerous clients accessing it. Dependency of the clients on such an interface imposes a constraint on the interface's changeability and extensibility, since changes to the interface (or sometimes even its implementation) have the potential to affect the large number of clients that are coupled to it. Further, when an abstraction has bloated implementation, it becomes difficult to figure out the places in the abstraction that need be modified to accommodate a change or enhancement.

- **Testability:** An abstraction with this smell poses considerable challenges to testing that abstraction. For instance, a complex method implementation with high Cyclomatic complexity requires a proportionate number of test cases to exercise all its paths (Note: Cyclomatic complexity of a method corresponds to the number of paths in that method, and hence the number of minimum test cases needed for testing that method).
- **Reliability:** It is also well known that complexity breeds defects. Thus, a class with complex methods is likely to harbor many runtime problems thus impacting it's reliability.

5.2.6 ALIASES

This smell is also known in the literature as:

- God class [28]: This smell occurs when a class has 50 or more methods or attributes.
- Fat interface [79]: This smell occurs when the interface provided by the class is not cohesive.
- Blob class [76]: This smell occurs when a class is very large and has a high complexity.
- Classes with complex control flow [59]: This smell occurs when a class has a very high Cyclomatic complexity.
- Too much responsibility [65]: This smell occurs when a class has "too much" responsibility.
- Local breakable (class) [77]: This smell occurs when a class has excessive responsibility and has many local dependencies.

5.2.7 PRACTICAL CONSIDERATIONS

Key classes

Key classes [52] abstract most important concepts in a system and tend to be large, complex, and coupled with many other classes in the system. While it is difficult to avoid such classes in real-world systems, it is still important to consider how they could be decomposed so that they become easier to maintain.

Auto-generated code

Often, classes that are created as part of automatically generated code have complex methods. For example, tools for generating parsers usually create complex code with extensive conditional logic. In practice, it is difficult to change the generator tool or manually refactor such complex code.

5.3 CYCLICALLY-DEPENDENT MODULARIZATION

This smell arises when two or more abstractions depend on each other directly or indirectly (creating a tight coupling between the abstractions).

5.3.1 RATIONALE

A cyclic dependency is formed when two or more abstractions have direct or indirect dependencies on each other. Cyclic dependencies between abstractions violate the Acyclic Dependencies Principle (ADP)[79] and Ordering Principle [80]. In the presence of cyclic dependencies, the abstractions that are cyclically-dependent may need to be understood, changed, used, tested, or reused together. Further, in case of cyclic dependencies, changes in one class (say A) may lead to changes in other classes in the cycle (say B). However, because of the cyclic nature, changes in B can have ripple effects on the class where the change originated (i.e., A). Large and indirect cyclic dependencies are usually difficult to detect in complex software systems and are a common source of subtle bugs. Since this smell is a result of not adhering to the enabling technique, "create acyclic dependencies," we name this smell Cyclically dependent Modularization.

A special form of cyclic dependency is sometimes exhibited within an inheritance hierarchy. A subtype has a dependency on its supertype because of the inheritance relationship. However, when the supertype also depends on the subtype (for instance, by having an explicit reference to the subtype), it results in a cyclic dependency. We refer to this special form of cyclic dependency as Cyclic Hierarchy smell, and discuss it in detail in Section 6.10.

UNDERSTANDING CYCLES

To better understand the impact of cyclic dependencies, let us first understand some terms related to cycles. A *dependency diagram* is a directed graph that shows the dependency relationship among abstractions. In a dependency diagram, if you start from one abstractions and reach the same abstraction by following one or more path(s) formed by the dependency edges, the abstractions in the followed path form a *cycle*. A *tangle* consists of more than one cycle. When abstractions are tightly coupled by a large number of direct or indirect cyclic dependencies, they form a *tangled design*, and the dependency graph looks unpleasantly complex.

Figure 5.10 shows a tangle between six abstractions in `java.util` package. In this graph, you can see cycles of various lengths. (It should be noted that the graph also shows Cyclic Hierarchy smell.)

In this design, any change to an abstraction involved in this dependency chain has the potential to affect other abstractions that depend on it, causing ripple effects or cascade of changes. A designer must, therefore, strive for designs that do not consist of tangles.

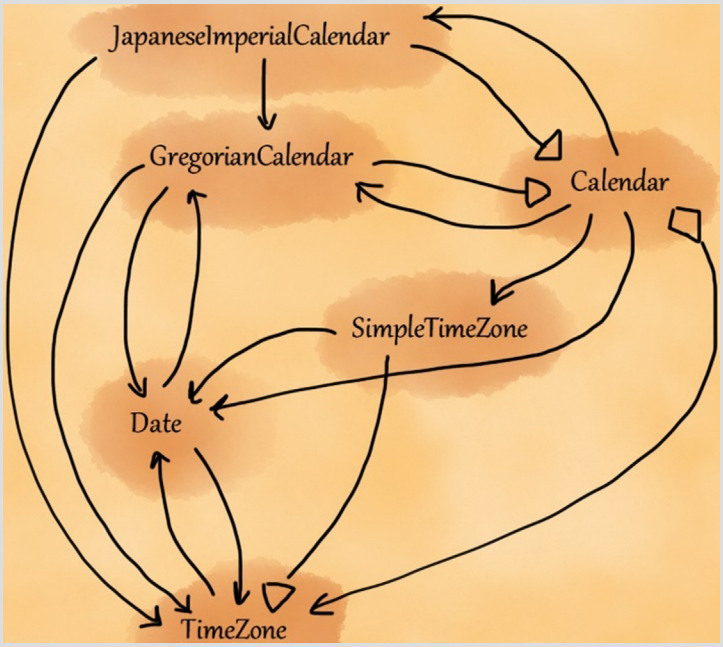

FIGURE 5.10

Cycles between six abstractions in java.util package.

5.3.2 POTENTIAL CAUSES

Improper responsibility realization

Often, when some of the members of an abstraction are wrongly misplaced in another abstraction, the members may refer to each other, resulting in a cyclic dependency between the abstractions.

Passing a self reference

A method invocation from one abstraction to another often involves data transfer. If instead of explicitly passing only the required data, the abstraction passes its own reference (for instance, via "this") to the method of another abstraction, a cyclic dependency is created.

Implementing call-back functionality

Circular dependencies are often unnecessarily introduced between two classes while implementing call-back[2] functionality. This is because inexperienced developers

[2]A call-back method is one that is passed as an argument to another method, and is invoked at a later point in time (for instance, when an event occurs).

may not be familiar with good solutions such as design patterns [54] which can help break the dependency cycle between the concerned classes.

Hard-to-visualize indirect dependencies

In complex software systems, designers usually find it difficult to mentally visualize dependency relationships between abstractions. As a result, designers may inadvertently end up creating cyclic dependencies between abstractions.

5.3.3 EXAMPLES

Example 1

Let us consider a storage application from the Cloud ecosystem. This application allows a client to upload its data to the Cloud where it can be archived. Since security and privacy are important concerns in the Cloud context, the client-side application encrypts the data before uploading it to the Cloud. Assume that this application consists of a class named `SecureDocument` that uses a `DESEncryption`[3] object to encrypt a document. To encrypt the contents of the document, the `SecureDocument` calls a method `encrypt()` provided in `DESEncryption` class and passes the "this" pointer as its argument. For instance, the call may look like this:

```
SecureDocument encryptedDocument = desEncryption.encrypt(this);
```

The method `encrypt` in `DESEncryption` is declared as follows:

```
SecureDocument encrypt(SecureDocument docToEncrypt)
```

Using the "this" pointer within the `encrypt` method, the `DESEncryption` object fetches the contents of the document, encrypts it, and returns an encrypted document object. Thus, `SecureDocument` and `DESEncryption` know about each other, leading to the Cyclically-dependent Modularization smell between them, as shown in Figure 5.11.

Example 2

Consider the case of a medical application that supports encryption of scanned images before storing them on a drive. This application consists of a `SecurityManager` class that fetches an encrypted image from an `Image` class (Figure 5.12). The `Image` class, in turn, uses an `Encryption` class to encrypt its contents; during this process, the `Image` class passes the "this" pointer to the `encrypt()` method within the `Encryption` class. When invoked, the `encrypt()` method within the `Encryption`

[3]DES stands for Data Encryption Standard; it is a well-known symmetric key algorithm for encrypting data.

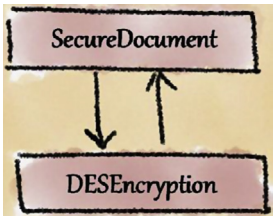

FIGURE 5.11

Cyclic dependency between SecureDocument and DESEncryption (Example 1).

FIGURE 5.12

Cyclic dependency between Image and Encryption classes (Example 2).

class fetches the `Image` contents, and returns the contents after encryption. Here, the `Image` and `Encryption` classes are dependent on each other, hence this design exhibits the Cyclically-dependent Modularization smell.

Example 3

Consider an order-processing module in an e-commerce application. In this application, assume that you have two classes named `Order` and `OrderImpl` that provide support for order processing (Figure 5.13). The `Order` class maintains information about an order and the associated information about the ordered items. The method `getAmount()` in `Order` class uses `computeAmount()` method of `OrderImpl` class. In

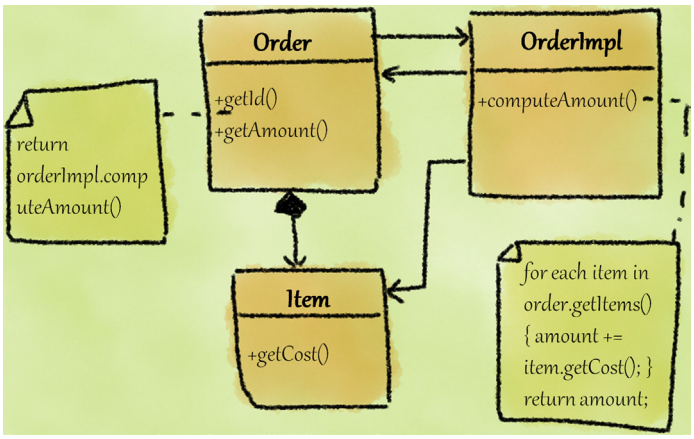

FIGURE 5.13

Cyclic dependency between Order and OrderImpl classes (Example 3).

turn, the `computeAmount()` method extracts all the items associated with the `Order` object and computes the sum of costs of all the items that are a part of the order. In effect, classes `Order` and `OrderImpl` depend on each other, hence the design has the Cyclically-dependent Modularization smell.

Example 4

Assume that an order-processing application has an `Order` class encapsulating information about an order such as `name`, `id`, and `amount`. The `getAmount()` method of the `Order` class uses `computeAmount()` method of `TaxCalculator` class. The `computeAmount()` method fetches all the items associated with the `Order` object and computes a summation of each item cost. In addition, it calls `calculateTax()` method, adds the computed tax with the running amount, and returns it. In this case, classes `Order` and `TaxCalculator` depend on each other, as shown in Figure 5.14. Clearly, the design fragment shows the Cyclically-dependent Modularization smell.

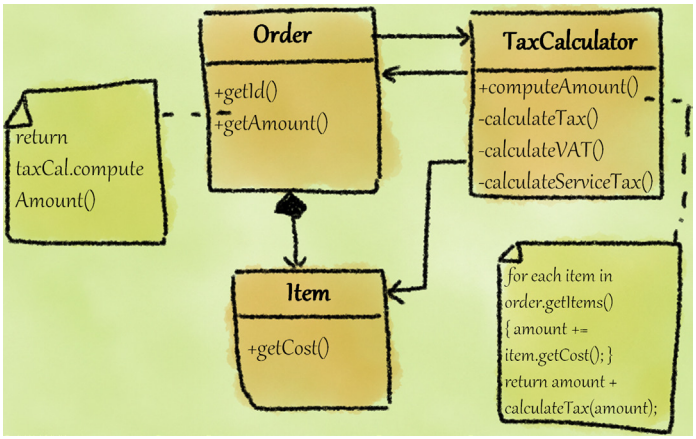

FIGURE 5.14

Cyclic dependency between Order and TaxCalculator classes (Example 4).

5.3.4 SUGGESTED REFACTORING

The refactoring for this smell involves breaking the dependency cycle. There are many strategies to do this; some important ones are:

- **Option 1:** Introduce an interface for one of the abstractions involved in the cycle.
- **Option 2:** In case one of the dependencies is unnecessary and can be safely removed, then remove that dependency. For instance, apply "move method" (and "move field") refactoring to move the code that introduces cyclic dependency to one of the participating abstractions.
- **Option 3:** Move the code that introduces cyclic dependency to an altogether different abstraction.

- **Option 4:** In case the abstractions involved in the cycle represent a semantically single object, merge the abstractions into a single abstraction.

As an illustration, consider the direct cyclic dependency between class A and class B (Figure 5.15). Figure 5.16, Figure 5.17, Figure 5.18, and Figure 5.19 respectively show the four options for refactoring to remove the cycle.

Suggested refactoring for Example 1

For the storage application from the Cloud ecosystem, a suggested refactoring is to introduce an interface `IEncryption` and have the class `DESEncryption` implement the interface. The `SecureDocument` class now depends on `IEncryption` interface instead of depending on the concrete `DESEncryption` class. This results in a design that removes the cyclic dependency. Interfaces are less likely to change than a concrete type, thus the resultant design is more stable than the original design with cyclic dependency. For instance, a

FIGURE 5.15

Cyclic dependency between classes A and B.

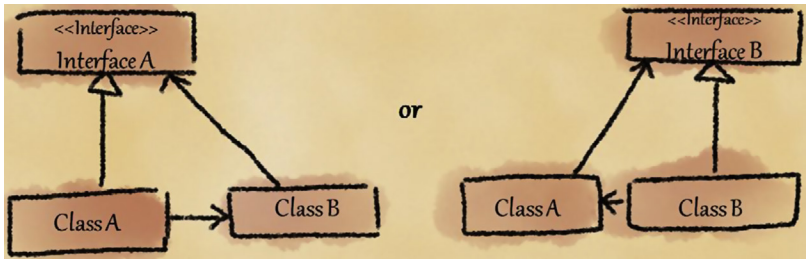

FIGURE 5.16

Breaking a cyclic dependency by introducing an interface (Option 1).

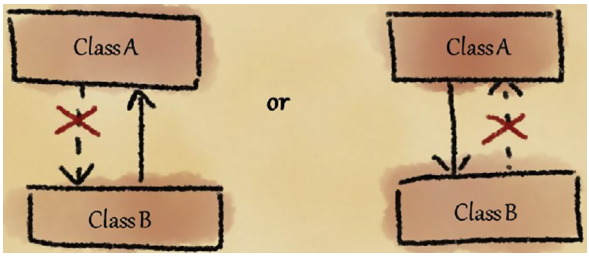

FIGURE 5.17

Breaking a cyclic dependency by removing a dependency (Option 2).

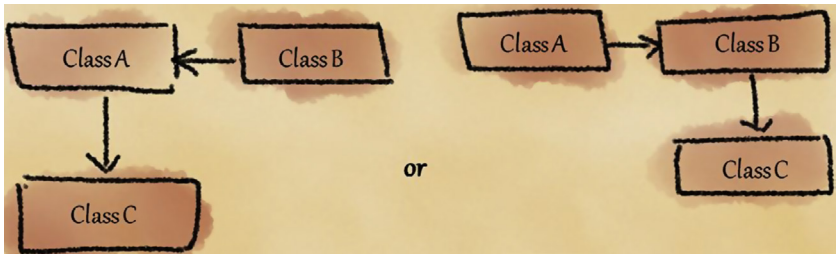

FIGURE 5.18

Breaking a cyclic dependency by introducing another abstraction (Option 3).

FIGURE 5.19

Breaking a cyclic dependency by merging the abstractions (Option 4).

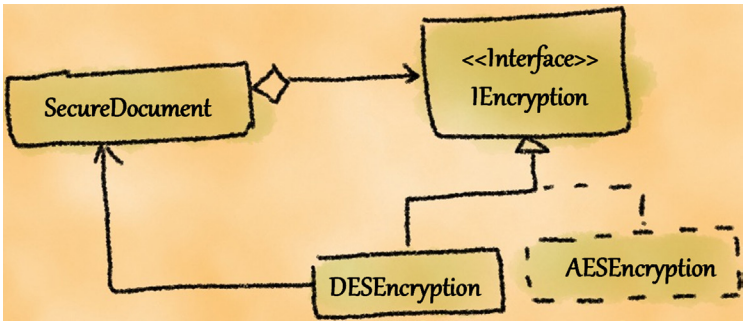

FIGURE 5.20

Suggested refactoring for cyclic dependency between SecureDocument and DESEncryption (Example 1).

new class such as `AESEncryption` (that supports the AES[4] encryption algorithm) can be added to the design without affecting the `SecureDocument` class (Figure 5.20).

Suggested refactoring for Example 2

For the healthcare application that requires image encryption, a suggested refactoring is to shift the responsibility of encrypting an image object from `Image` class to `SecurityManager` class to break the cycle. In the refactored design, `SecurityManager` class depends on `Image` and `Encryption` classes. `SecurityManger` class directly invokes image encryption on `Encryption` class by providing the

[4] Advanced Encryption Standard.

`Image` object. The `Encryption` class, in turn, uses the `Image` object reference provided by the `SecurityManager` to fetch the content of the image object, encrypt the content, and return the encrypted content to `SecurityManager`. Here, by removing the dependency from `Image` class to `Encryption` class, the cyclic dependency is broken (Figure 5.21).

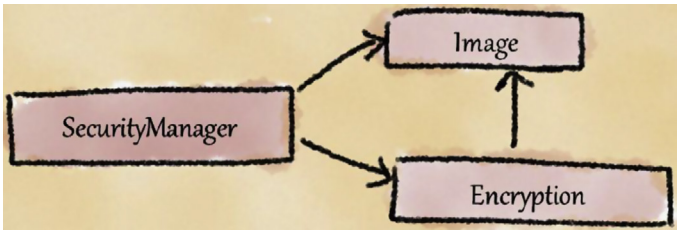

FIGURE 5.21

Suggested refactoring for cyclic dependency between Image and Encryption (Example 2).

Suggested refactoring for Example 3

In the case of the `Order` example, classes `Order` and `OrderImpl` are tightly coupled to each other, and they semantically represent an order abstraction. Since the `OrderImpl` class has only one method, this class could be merged into the `Order` class. When we apply this refactoring, the cyclic dependency disappears, since there is only one class (Figure 5.22).

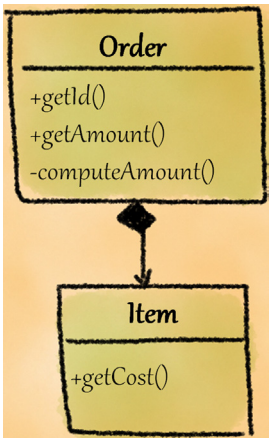

FIGURE 5.22

Suggested refactoring for cyclic dependency in order-processing module (Example 3).

Suggested refactoring for Example 4

To refactor the `Order` example with the `TaxCalculator` class, we can employ "move method" refactoring. The `computeAmount()` originally belonging to `TaxCalculator` class can be moved to `Order` class. By moving this method, we eliminate the dependency from `TaxCalculator` to `Order` class (Figure 5.23).

ANECDOTE

One of the authors was working as a software design consultant for a startup company. The product that the company was developing was originally designed by an experienced architect, but he had recently quit the company. Most developers in the team were fresh engineers, a company strategy intended to keep costs low until they could get further funding.

With lack of experienced designers or architects, development proceeded without any focus on architecture or design quality. The management prided itself on its pragmatic view and pushed the development team to meet functional requirements and get the product to the market on time.

In this process, the author noticed that numerous compromises were made in the design. For instance, the original design was a layered architecture with strict separation of concerns. However, with no architect or designer to oversee the development, developers started introducing business logic in UI classes such as `Buttons` and `Panels`. Worse, low-level utility classes such as `GraphUtilities` also directly referred to GUI classes. These aspects clearly indicate violation of the layering style.

To demonstrate the extent of the problems to management, the author ran dependency analysis tools and found innumerable dependency cycles across layers. He also used visualization tools to demonstrate how the whole codebase was excessively tangled. It was evident that touching code in any class could potentially break the working software. Due to the extent of the problem and the impending release, management could not take immediate remedial steps.

While the product was well received by the customers when it was launched, the design was so brittle that even small changes would break the working software. Upon the recommendation of the author, the management recruited an experienced designer as the architect. All further development was frozen, and considerable refactoring was performed to clean up the design. After a few months of refactoring, the design quality significantly improved to an extent that development tasks could commence. However, because the technical debt had to be repaid through extensive refactoring, the second release was delayed considerably from its originally scheduled date.

Key take-aways from this experience are:

- It is important to have a designated architect who actively ensures that architectural erosion or design decay does not happen.
- If the technical debt is not repaid through periodic refactoring, it will eventually stall the project.

5.3.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability:** Since all the abstractions that are cyclically-dependent can be understood only together, it impacts the understandability of the design.
- **Changeability** and **Extensibility:** Making changes to an abstraction that is part of a cycle can cause ripple effects across classes in the dependency chain (including the original abstraction). This makes it difficult to understand, analyze, and implement new features or changes to any abstraction that is part of a

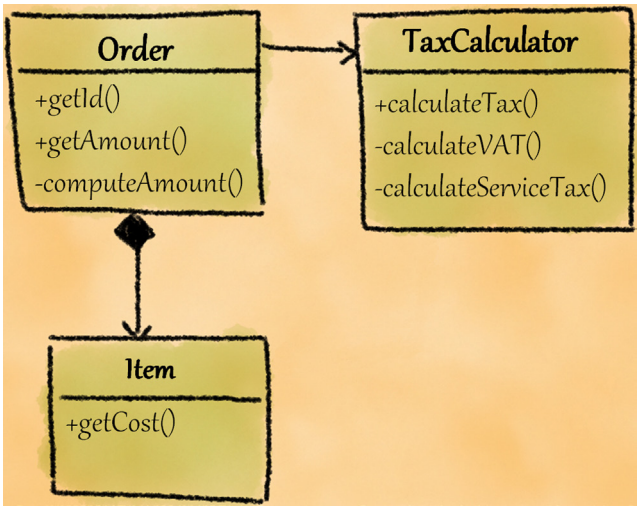

FIGURE 5.23

Suggested refactoring for cyclic dependency between Order and TaxCalculator (Example 4).

dependency cycle. Hence, this smell impacts the changeability and extensibility of the design.

- **Reusability:** The abstractions in a dependency cycle can only be reused together. Hence, this smell impacts reusability of the design.
- **Testability:** Since it is difficult to independently test the abstractions participating in the cycle, this smell impacts testability.
- **Reliability:** In a dependency chain, changes to any abstraction can potentially manifest as runtime problems across other abstractions. For instance, consider the case where a value is represented as a double in a class and that the class is part of a long cycle. If the value is changed to float type in the class, the impact of this change on other classes may not be evident at compile-time; however, it is possible that it manifests as a floating-point error at runtime in other classes.

5.3.6 ALIASES

This smell is also known in literature as:

- Dependency cycles [72, 73, 74]: This smell occurs when a class has circular references.
- Cyclic dependencies [71, 75, 76]: This smell occurs when classes are tightly coupled and mutually dependent.
- Cycles [9]: This smell occurs when one of the (directly or indirectly) used classes is the class itself.

- Bidirectional relation [57]: This smell occurs when two-way dependencies between methods of two classes are present.
- Cyclic class relationships [70]: This smell occurs when classes have improper or questionable relationships to other classes, such as codependence.

5.3.7 PRACTICAL CONSIDERATIONS

Unit cycles between conceptually related abstractions

Cycles of size one (i.e., cycles that consist of exactly two abstractions) are known as *unit cycles*. Often, unit cycles are formed between conceptually related pairs. For instance, consider the classes `Matcher` and `Pattern` (both part of `java.util.regex` package) that are cyclically-dependent on each other (see Figure 5.24). These two classes are almost always understood, used, or reused together. In real-world projects, it is common to find such unit cycles. However, since the unit cycle is small, it is easier to understand, analyze, and implement changes within the two abstractions that are part of the cycle.

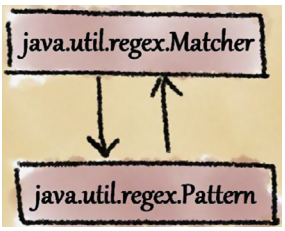

FIGURE 5.24

Cyclic dependency between Matcher and Pattern.

However, large cycles and tangles (which are commonly observed in large real-world software systems) make it considerably difficult to maintain the software. It is, therefore, important for software designers in real-world projects to focus on reducing cyclic dependencies between abstractions.

5.4 HUB-LIKE MODULARIZATION

This smell arises when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions.

5.4.1 RATIONALE

"High cohesion and low coupling" is the basis for effective modularization. Meyer's "few interfaces" rule for modularity says that "every module should communicate with as few others as possible." [24] For effective modularization, we must follow

the enabling technique "limit dependencies." When an abstraction has large number of incoming and outgoing dependencies, the principle of modularization is violated.

When an abstraction has a large number of incoming dependencies and outgoing dependencies, the dependency structure looks like a hub, hence we name this smell Hub-like Modularization.

5.4.2 POTENTIAL CAUSES

Improper responsibility assignment (to a hub class)

When an abstraction is overloaded with too many responsibilities, it tends to become a hub class with a large number of incoming as well as outgoing dependencies. In other words, most classes in the design will talk to this hub class, and this hub class also communicates with most other classes.

"Dump-all" utility classes

Classes that provide supporting functionality often grow very large and get coupled with a large number of other classes. Consider the `javax.swing.SwingUtilities` class that is used by numerous other classes in the swing package. This is not surprising given the fact that it is a utility class. However, what is surprising is the number of UI classes in swing package it refers back to—such a large number of incoming and outgoing dependencies makes this class a hub class. In our experience, we have seen that utility classes often become common "dump-all-ancillary-functionality-here" classes that turn out to be hub classes.

5.4.3 EXAMPLES

Example 1

The `java.awt.Font` class represents fonts and supports functionality to render text as a sequence of glyphs on `Graphics` and `Component` objects. This class has 19 incoming dependencies and 34 outgoing dependencies, as shown in Figure 5.25. Note that these do not include dependencies from method implementations within this class.

Example 2

`java.util.Component` class abstracts graphical objects that can be displayed on the screen, such as buttons and scrollbars. This class has 498 incoming dependencies and 71 outgoing dependencies, as shown in Figure 5.26. Note that these do not include dependencies from method implementations within this class.

5.4.4 SUGGESTED REFACTORING

A refactoring solution for this smell may involve applying one or more of the following:

- If the hub class has multiple responsibilities, indicating improper responsibility assignment, the refactoring suggestion is to split up the responsibilities across multiple new/old abstractions so that the number of incoming and outgoing dependencies is reduced.

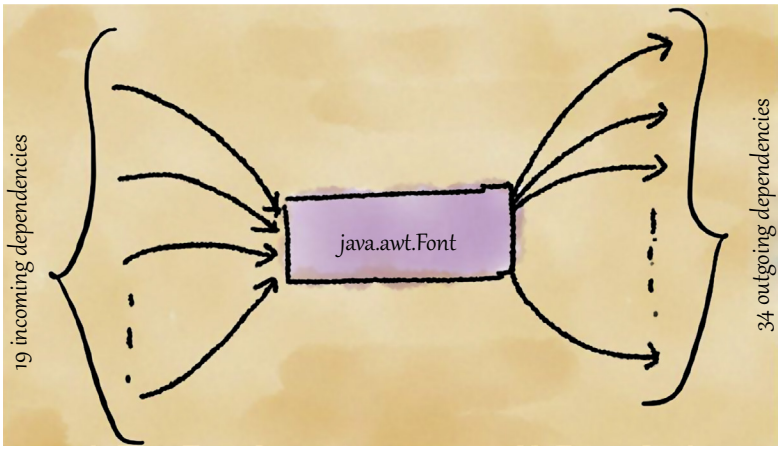

FIGURE 5.25

Incoming and outgoing dependencies of java.awt.Font class (Example 1).

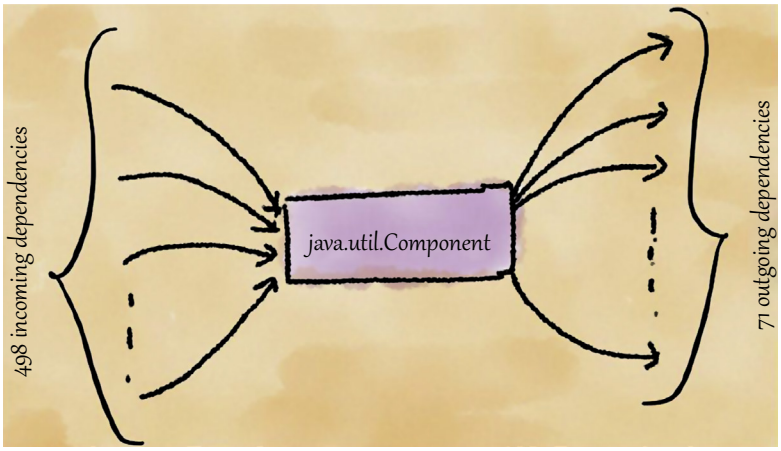

FIGURE 5.26

Incoming and outgoing dependencies of java.awt.Component class (Example 2).

- If the dependencies are caused due to misplaced members in the hub class, the refactoring suggestion is to assign those members to appropriate abstractions.
- Sometimes the Chain of Responsibility [54] pattern can be used to reduce the number of incoming dependencies on a hub class. For example, in the scenario where a number of candidate receivers are interested in value or data held by the hub class, the number of incoming dependencies on the hub class will be high. To address this, one possible refactoring is to connect all the receivers via a Chain of Responsibility and have the hub class notify only the first receiver in the chain about the change in value. The receiver would, in

turn, cascade the notification to other receivers in the chain. With this refactoring, the number of incoming dependencies on the hub class drastically reduces.

Refactoring for Example 1

The `java.awt.Font` class has some members that could be assigned to more suitable classes. For instance, consider the overloaded versions of the `createGlyphVector()`

```
public GlyphVector createGlyphVector(FontRenderContext frc.
CharacterIterator ci) {
  return (GlyphVector)new StandardGlyphVector(this, ci, frc);
}
```

methods; see one definition of the method below:

The method definition of `createGlyphVector()` shows that the `Font` class has direct knowledge of `GlyphVector` class. Although, conceptually, glyphs are related to fonts, glyph-related functionality need not belong to `Font` class, and could be moved to `GlyphVector` class. Now, any client code that originally depended on the `Font` class for glyph related functionality can directly use the `GlyphVector` class. Such refactorings can help reduce some of the incoming as well as outgoing dependencies of the `Font` class.

Refactoring for Example 2

Considering the size, complexity, and number of dependencies of `java.awt.Component` class, different kinds of refactorings (such as move method, split class, extract class, and extract method) may be performed. To illustrate how a "move method" refactoring can be applied, consider the following public methods of this class:

```
FontMetrics getFontMetrics(Font font)
Image createImage(ImageProducer producer)
Image createImage(int width, int height)
VolatileImage createVolatileImage(int width, int height)
VolatileImage createVolatileImage(int width, int height,
ImageCapabilities caps)
```

It can be argued that these are ancillary or supportive methods that do not belong to `Component` class. If we move these methods to relevant classes such as `FontMetrics`, `Image`, and `VolatileImage`, the related dependencies would also move to these classes. This, in turn, would help reduce the number of incoming and outgoing dependencies of the `Component` class.

5.4.5 IMPACTED QUALITY ATTRIBUTES

- **Understandability:** For understanding a hub class, one may have to look up and understand many other classes that the class depends on. Further, the presence of a hub class may make it difficult to understand dependencies among various classes. These factors impact understandability of the design.
- **Changeability, Extensibility,** and **Reliability:** When an abstraction is depended upon by numerous other abstractions, any modification to that abstraction has the potential to affect all the other abstractions that depend on it. For this reason, modifying a hub class is difficult. Similarly, when an abstraction depends on numerous other abstractions, it is subject to ripple effects from the modifications to any of these abstractions. For this reason, a hub class can be affected by numerous other abstractions, and can in turn affect abstractions that depend on that hub class. In other words, designs with hub classes are prone to ripple effects of changes done to the design, impacting the changeability and extensibility of the design. Sometimes (as previously discussed in the case of Cyclically-dependent Modularization), these ripple effects may manifest as runtime problems, impacting reliability.
- **Reusability** and **Testability:** Given the large number of outgoing dependencies that the hub class has with other classes in the design, it is difficult to decouple the hub class to reuse it in other contexts or test it independently.

5.4.6 ALIASES

This smell is also known in literature as:

- Bottlenecks [55, 71]: This smell occurs when a class refers to many other classes and is used by many other classes.
- Local hubs [77]: This smell occurs when a type has many immediate dependencies and many immediate dependents.
- Man-in-the-middle [25]: This smell occurs when the design has a central class that serves as a kind of a mediator for many other classes or even other modules.

5.4.7 PRACTICAL CONSIDERATIONS

Core abstractions

If you analyze large object oriented applications, you will find that there are a few "core abstractions" that play a central role in the design. From our experience, we find that it is usually difficult to keep the number of incoming as well as outgoing dependencies low for such core abstractions. For instance, consider the class `java.lang.Class` in JDK. This class represents all classes and interfaces in a Java application, and hence is a core abstraction in JDK. Since the `java.lang.Class` has more than 1000 incoming dependencies and 40 outgoing dependencies, it has Hub-like Modularization smell. However, limiting the number of incoming and outgoing dependencies of such core abstractions is difficult (if not impossible).