Chapter 3. Modularity

Architects and developers have struggled with the concept of modularity for quite some time, as is evident in this quote from *Composite/Structured Design* (Van Nostrand Reinhold, 1978):

> 95% of the words [written about software architecture] are spent extolling the benefits of "modularity" and little, if anything, is said about how to achieve it.
>
> Glenford J. Myers

Different platforms offer different reuse mechanisms for code, but all support some way of grouping related code together into *modules*. While this concept is universal in software architecture, it has proven slippery to define. A casual internet search yields dozens of definitions, with no consistency (and some contradictions). This isn't a new problem. However, because no recognized definition exists, we must jump into the fray and provide our own definitions for the sake of consistency throughout the book.

Understanding modularity and its many incarnations in the development platform of choice is critical for architects. Many of the tools we have to analyze architecture (such as metrics, fitness functions, and visualizations) rely on modularity and related concepts. *Modularity* is an organizing principle. If an architect designs a system without paying attention to how the pieces wire together, that system will present myriad difficulties. To use a physics analogy, software systems model complex systems, which tend toward entropy (or disorder). In a physical system, energy must be added to preserve order. The same is true for software systems: architects must constantly expend energy to ensure structural soundness, which won't happen by accident.

Preserving good modularity exemplifies our definition of an *implicit* architecture characteristic: virtually no project requirements explicitly ask the architect to ensure good modular distinction and communication, but sustainable code bases do require the order and consistency that this brings.

Modularity Versus Granularity

Developers and architects often use the terms *modularity* and *granularity* interchangeably, yet their meanings are very different. Modularity is about breaking systems apart into smaller pieces, such as moving from a monolithic architecture style (like the traditional n-tiered layered architecture) to a highly distributed architecture style, like microservices. Granularity, on the other hand, is about the *size* of those pieces—how big a particular part of the system (or service) should be. However, as stated in the following quote by one of your authors, it's with *granularity* where architects and developers get into trouble:

> Embrace modularity, but beware of granularity.
>
> Mark Richards

Granularity causes services or components to be coupled to one another, creating complex, hard-to-maintain architecture antipatterns like *Spaghetti Architecture*, *Distributed Monoliths*, and the famous *Big Ball of Distributed Mud*. The trick to avoiding these architectural antipatterns is to pay attention to granularity and the overall level of coupling between services and components.

Defining Modularity

Merriam-Webster defines a *module* as "each of a set of standardized parts or independent units that can be used to construct a more complex structure." By contrast, in this book, we use *modularity* to describe a logical grouping of related code, which could be a group of classes in an object-oriented language or a group of functions in a structured or functional language. Developers typically use modules as a way to group related code together. For example, the `com.mycompany.customer` package in Java should contain things related to customers. Most languages provide mechanisms for modularity (`package` in Java, `namespace` in .NET, and so on).

Modern programming languages feature a wide variety of packaging mechanisms, and many developers find it difficult to choose between them. For example, in many modern languages, developers can define behavior in functions/methods, classes, or packages/namespaces, each with different visibility and scoping rules. Some languages complicate this further by adding programming constructs, such as the metaobject protocol, to provide even more extension mechanisms.

Architects must be aware of how developers package things, because packaging has important implications in architecture. For example, if several packages are tightly coupled, reusing one of them for related work becomes more difficult.

Modular Reuse Before Classes

Developers who trained in the days before object-oriented languages may puzzle over why there are so many different separation schemes. Much of the reason has to do with backward compatibility—not of code, but of how developers think about things.

In March 1968, the journal *Communications of the Association for Computing Machinery (ACM)* published a paper from computer scientist Edsger Dijkstra entitled "Go To Statement Considered Harmful." He denigrated the common use of the `GOTO` statement, common in programming languages at the time, because it allowed nonlinear leaping around within code, making reasoning and debugging difficult.

Dijkstra's paper helped usher in the era of *structured* programming languages in the mid-1970s, exemplified by Pascal and C, which encourage deeper thinking about how things fit together. Developers quickly realized that most programming languages offered no good way to group like things together logically. Thus, the short era of *modular* languages was born in the mid-1980s, such as Modula (Pascal creator Niklaus Wirth's next language) and Ada. These languages embraced the programming construct of the *module*, much as we think about packages or namespaces today (but without the classes).

However, the modular programming era of the mid-1980s was short-lived because object-oriented languages became popular and offered new ways to encapsulate and reuse code. Still, language designers realized the utility of modules and retained them in the form of packages and namespaces. Many languages still contain compatibility features that seem odd today but were introduced to support these different paradigms. For example, Java supports modular paradigms (via packages and package-level initialization using static initializers), as well as object-oriented and functional paradigms, each with its own scoping rules and quirks.

In this book's discussions about architecture, we use *modularity* as a general term to denote a related grouping of code: classes, functions, or any other grouping. This doesn't imply a physical separation, merely a logical one. (The difference is sometimes important.) For example, lumping a large number of classes together in a monolithic application may be convenient; however, when it comes time to restructure the architecture, the coupling encouraged by loose partitioning can impede efforts to break the monolith apart. That's why it's useful to talk about modularity as a concept, separate from the physical separation that a particular platform forces or implies.

It is worth discussing the general concept of a *namespace*, which is separate from the technical implementation in the .NET platform that is also called a namespace. Developers often need precise, fully qualified names for different software assets (components, classes, and so on) to separate them from each other. The most obvious example that people use every day is the internet, which relies on unique, global identifiers tied to IP addresses.

Most languages have some modularity mechanism that doubles as a namespace to organize things like variables, functions, or methods. Sometimes the module structure is reflected physically: Java package structures, for example, must reflect the directory structure of the physical class files.

A Language with No Name Conflicts: Java 1.0

The original designers of Java had extensive experience dealing with name conflicts and clashes, which were common in programming platforms at the time. Java 1.0 used a clever hack to avoid ambiguity when two classes had the same name—such as if the problem domain included a catalog *order* and an installation *order*, both named *order* but with very different connotations (and classes). The Java designers' solution was to create the `package` namespace mechanism, along with a requirement that the physical directory structure must match the package name. Because filesystems won't allow two files with the same name to reside in the same directory, this leveraged the operating system's inherent features to ambiguity and name conflicts. Thus, the original `classpath` in Java contained only directories.

However, as the language designers discovered, forcing every project to have a fully formed directory structure was cumbersome, especially as projects became larger. Plus, building reusable assets was difficult: frameworks and libraries had to be "exploded" into the directory structure. In the second major release of Java (1.2, but called Java 2), designers added the `jar` mechanism, which allows an archive file to act as a directory structure on a classpath. For the next decade, Java developers struggled with getting the classpath exactly right, as a combination of directories and JAR files. Their original intent had been broken: now two JAR files *could* create conflicting names on a classpath. This is why Java developers of that era tend to have numerous war stories about debugging class loaders.

Measuring Modularity

Since modularity is so important, architects need tools to help them better understand it. Fortunately, researchers have created a variety of language-agnostic metrics for this purpose. We'll focus here on three key concepts: *cohesion*, *coupling*, and *connascence*.

Cohesion

Cohesion refers to the extent to which a module's parts should be contained within the same module. In other words, it measures how related the parts are to one another. An ideal cohesive module is one where all parts are packaged together; breaking them into smaller pieces would require coupling the parts together via calls between modules to achieve useful results. The cautionary tale of modularity as it relates to cohesion is exemplified in the following quote from the book *Structured Design* (Pearson, 2008):

> Attempting to divide a cohesive module would only result in increased coupling and decreased readability.
>
> Larry Constantine

Computer scientists have defined a range of cohesion, measured from best to worst:

Functional cohesion

> Every part of the module is related to the other, and the module contains everything essential it needs to function.

Sequential cohesion

> Two modules interact: one outputs data that becomes the input for the other.

Communicational cohesion

> Two modules form a communication chain in which each operates on information and/or contributes to some output. For example, one adds a record to the database and the other generates an email based on that information.

Procedural cohesion

> Two modules must execute code in a particular order.

Temporal cohesion

> Modules are related based on timing dependencies. For example, many systems have a list of seemingly unrelated things that must be initialized at system startup; these different tasks are *temporally cohesive*.

Logical cohesion

> The data within modules is related logically but not functionally. For example, consider a module that converts information from text, serialized objects, or streams into some other format. Its operations are related, but the functions are quite different. A common example of this type of cohesion

exists in virtually every Java project in the form of the `StringUtils` package, a group of static methods that operate on `String` but are otherwise unrelated.

Coincidental cohesion

: The elements in a module are unrelated other than being in the same source file. This represents the most negative form of cohesion.

Despite its many variants, *cohesion* is a less precise metric than *coupling*. Often, a particular module's degree of cohesion is determined at the discretion of a particular architect. Consider this module definition:

`Customer Maintenance`

- `add customer`
- `update customer`
- `get customer`
- `notify customer`
- `get customer orders`
- `cancel customer orders`

Should the last two entries reside in this module? Or should the developer create two separate modules? Here's what that would look like:

`Customer Maintenance`

- `add customer`
- `update customer`
- `get customer`
- `notify customer`

`Order Maintenance`

- `get customer orders`
- `cancel customer orders`

Which is the correct structure? As always, it depends:

- Are these the only two operations for `Order Maintenance`? If so, it may make sense to collapse those operations back into `Customer Maintenance`.
- Is `Customer Maintenance` expected to grow much larger? If so, perhaps developers should look for opportunities to extract behavior into a different (or new) module.
- Does `Order Maintenance` require so much knowledge of `Customer` information that separating the two modules would require a high degree of coupling to make it functional? (This relates back to the prior Larry Constantine quote.)

These questions represent the kind of trade-off analysis that lies at the heart of a software architect's job.

Computer scientists have developed a good structural metric to determine cohesion—specifically, *Lack of Cohesion* (which is a bit surprising, given how subjective this characteristic is). A well-known set of metrics named the Chidamber and Kemerer Object-Oriented Metrics Suite measures particular aspects of object-oriented software systems. It includes many common code metrics, such as Cyclomatic Complexity (see “Cyclomatic Complexity”) and several important coupling metrics, discussed in “Coupling”.

Chidamber and Kemerer have also developed a Lack of Cohesion in Methods (LCOM) metric, which measures the structural cohesion of a module. The initial version appears in Equation 3-1.

Equation 3-1. LCOM, version 1

$$LCOM = \begin{matrix} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{matrix}$$

$$LCOM = \begin{matrix} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{matrix}$$

In this equation, *P* increases by 1 for any method that doesn’t access a particular shared field; *Q* decreases by 1 for methods that *do* share a particular shared field. If you find this formulation confusing, we sympathize—and it’s gradually become even more elaborate. The second variation, introduced in 1996 (thus the name *LCOM96B*), appears in Equation 3-2.

Equation 3-2. LCOM96B

$$LCOM96b = \frac{1}{a} \sum_{j=1}^{a} \frac{m - \mu(Aj)}{m}$$

$$LCOM96b = \frac{1}{a} \sum_{j=1}^{a} \frac{m - \mu(Aj)}{m}$$

We won’t bother untangling the variables and operators in Equation 3-2 because the following written explanation is clearer. Basically, the LCOM metric exposes incidental coupling within classes. A better definition of LCOM would be “the sum of sets of methods not shared via sharing fields.”

Consider a class with private fields `a` and `b`. Many of the methods only access `a`, and many other methods only access `b`. The *sum* of the sets of methods not shared via sharing fields (`a` and `b`) is high; therefore, this class incurs a high LCOM score, indicating a significant *lack of cohesion in methods*.

Consider the three classes shown in Figure 3-1. Here, fields appear as single letters inside octagons, and methods appear as blocks. In Class X, the LCOM score is low, indicating good structural cohesion. Class Y, however, lacks cohesion; each of the field/method pairs in Class Y could appear in its own class without affecting the system’s behavior. Class Z shows mixed cohesion; the last field/method combination could be refactored into its own class.

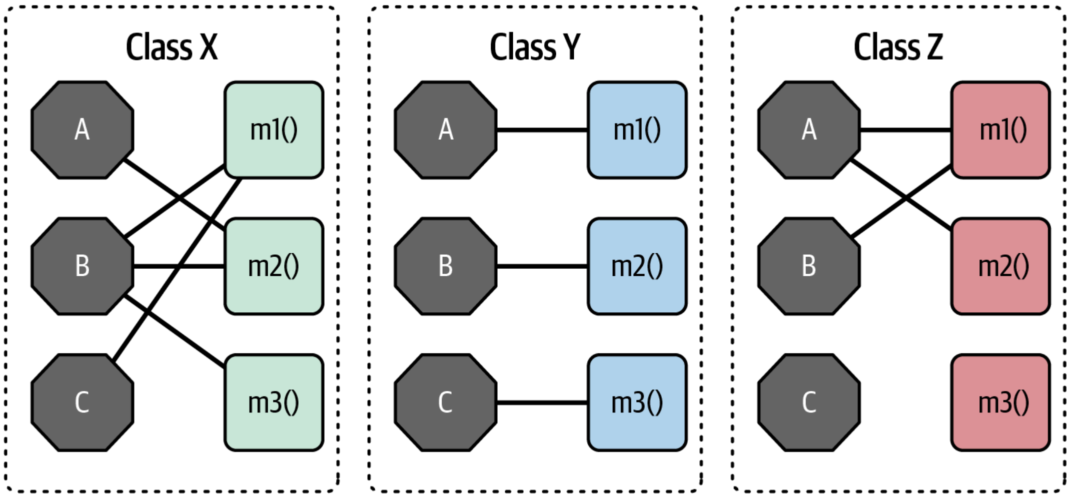

Figure 3-1. The LCOM metric, where fields are octagons, and methods are squares

The LCOM metric is useful to architects who are analyzing code bases in order to assist in restructuring, migrating, or understanding a code base. Shared utility classes are a common headache when moving architectures. Using the LCOM metric can help architects find classes that are incidentally coupled and should never have been a single class to begin with.

Many software metrics have serious deficiencies, and LCOM is not immune. All this metric can find is *structural* lack of cohesion; it has no way to determine if particular pieces fit together logically. This reflects back on our Second Law of Software Architecture: *why* is more important than *how*.

Coupling

Fortunately, we have better tools to analyze coupling in code bases. These are based in part on graph theory: because the method calls and returns form a call graph, it can be analyzed mathematically. Edward Yourdon and Larry Constantine's book *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Prentice-Hall, 1979), which defined many core concepts, including the metrics *afferent coupling* and *efferent coupling*. *Afferent* coupling measures the number of *incoming* connections to a code artifact (component, class, function, and so on). *Efferent* coupling measures the *outgoing* connections to other code artifacts. There are tools for virtually every platform that allow architects to analyze the coupling characteristics of code.

Why Such Similar Names for Coupling Metrics?

Why are two critical metrics in the architecture world that represent *opposite concepts* named virtually the same thing, differing in only the vowels that sound the most alike? These terms originate from the book *Structured Design*. Borrowing concepts from mathematics, Yourdon and Constantine coined the now-common terms *afferent* and *efferent* coupling. They should really have been called *incoming* and *outgoing* coupling, but the authors leaned toward mathematical symmetry rather than clarity. Developers have come up with several mnemonics to help out. For instance, *a* appears before *e* in the English alphabet, just as *incoming* appears before *outgoing*. The letter *e* in *efferent* matches the initial letter in *exit*, which helps in remembering that it represents outgoing connections.

Core Metrics

While component coupling has raw value to architects, several other derived metrics allow for deeper evaluation. The metrics discussed in this section were created by software engineer Robert C. Martin, and apply widely to most object-oriented languages.

Abstractness is the ratio of abstract artifacts (abstract classes, interfaces, and so on) to concrete artifacts (implementations). The Abstractness metric measures a code base's degree of abstractness versus implementation. For example, on one end of the scale would be a code base with no abstractions, just a huge, single function of code (as in a single `main()` method). The other end of the scale would be a code base with too many abstractions, making it difficult for developers to understand how things wire together. (For example, it takes developers a while to figure out what to do with an `AbstractSingletonProxyFactoryBean` abstract class, given its many layers of abstraction and ambiguous name.)

The formula for Abstractness appears in Equation 3-3.

Equation 3-3. Abstractness

$$A = \frac{\sum m^a}{\sum m^c + \sum m^a}$$

$$A = \frac{\sum m^a}{\sum m^c + \sum m^a}$$

Architects calculate Abstractness by calculating the ratio of the sum of abstract artifacts to the sum of the concrete and abstract ones. In the equation, $m^a m^a$ represents *abstract* elements (interfaces or abstract classes) with the module, and $m^c m^c$ represents *concrete* elements (nonabstract classes). This metric looks for the same criteria. The easiest way to visualize this metric is to consider an application with 5,000 lines of code, all in one `main()` method. Its Abstractness numerator would be 1, while the denominator would be 5,000, yielding an Abstractness score of almost 0. That's how this metric measures the ratio of abstractions in code.

Another derived metric, *Instability*, is defined as the ratio of efferent coupling to the sum of both efferent and afferent coupling, as shown in Equation 3-4.

Equation 3-4. Instability

$$I = \frac{C^e}{C^e + C^a}$$

$$I = \frac{C^e}{C^e + C^a}$$

In the equation, $c^e c^e$ represents efferent (or outgoing) coupling, and $c^a c^a$ represents afferent (or incoming) coupling.

The Instability metric determines the *volatility* of a code base. A code base that exhibits high degrees of instability breaks more easily when changed because of high coupling. For example, if a class calls too many other classes to delegate work, the calling class will show high susceptibility to breakage if one or more of the called methods changes.

Distance from the Main Sequence

One of the few holistic metrics architects have for architectural structure is *Distance from the Main Sequence*, a derived metric based on Instability and Abstractness, shown in Equation 3-5.

Equation 3-5. Distance from the Main Sequence

$$D = |A + I - 1|$$

$$D = | A + I - 1|$$

In the equation, AA = Abstractness and II = Instability.

Note that both Abstractness and Instability are fractions whose results will always fall between 0 and 1 (except in some extreme cases). Thus, graphing the relationship produces the graph in Figure 3-2.

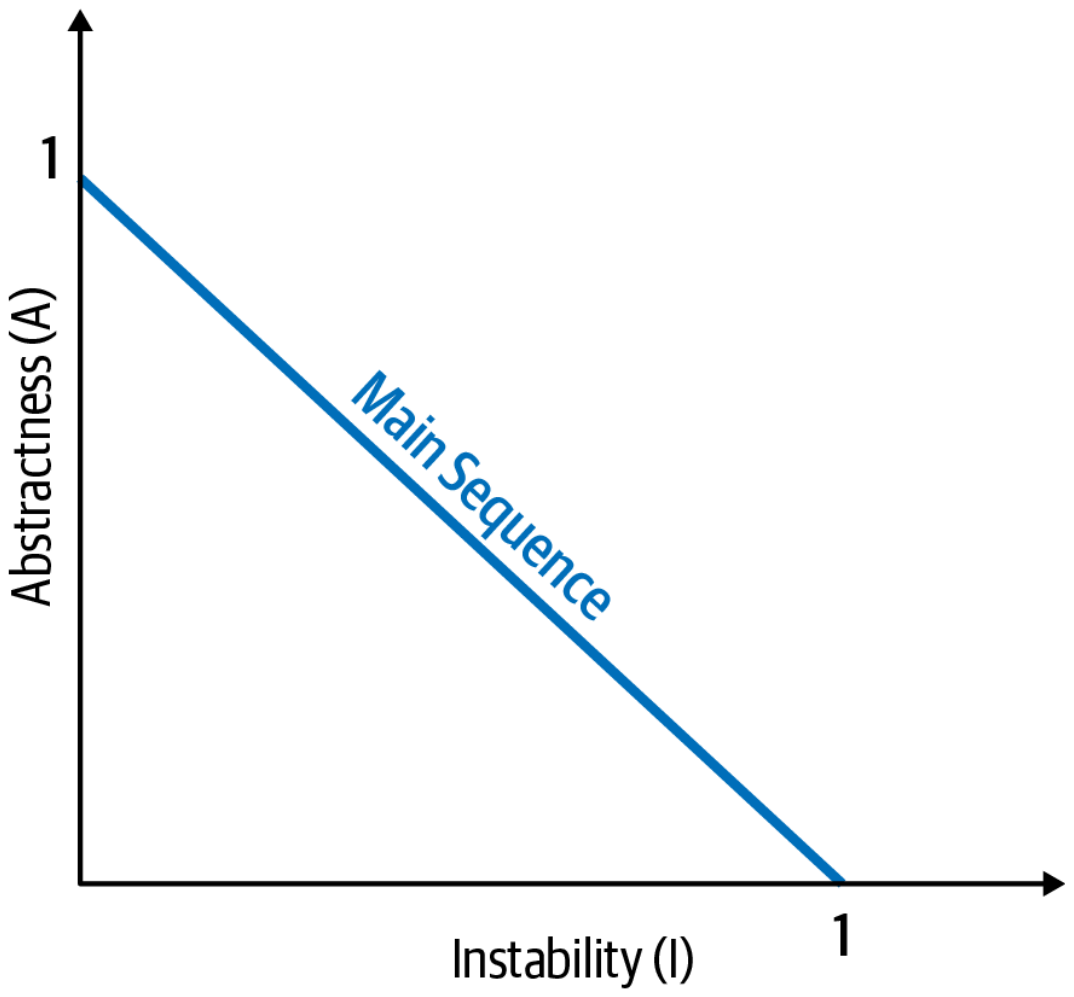

Figure 3-2. The main sequence defines the ideal relationship between Abstractness and Instability

The *Distance* metric imagines an ideal relationship between Abstractness and Instability; classes that fall near this idealized line exhibit a healthy mixture of these two competing concerns. For example, graphing a particular class allows developers to calculate the *Distance from the Main Sequence* metric, illustrated in Figure 3-3.

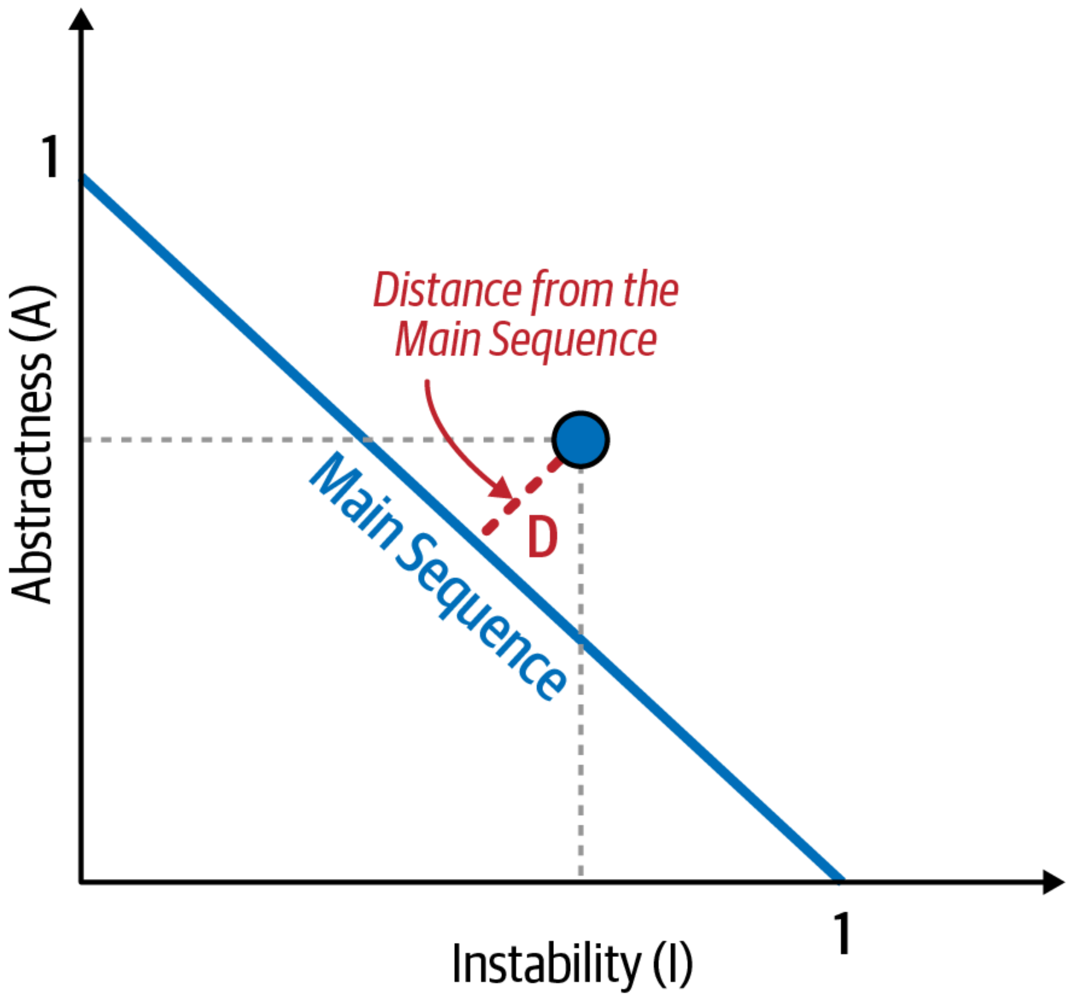

Figure 3-3. Normalized Distance from the Main Sequence for a particular class

The Figure 3-3 metric graphs the candidate class, then measures its distance from the idealized line. The closer to the line, the better balanced the class. Classes that fall too far into the upper righthand corner enter what architects call the *Zone of Uselessness*: code that is too abstract becomes difficult to use. Conversely, code that falls into the lower lefthand corner, as illustrated in Figure 3-4, enters the *Zone of Pain*: code with too much implementation and not enough abstraction becomes brittle and hard to maintain.

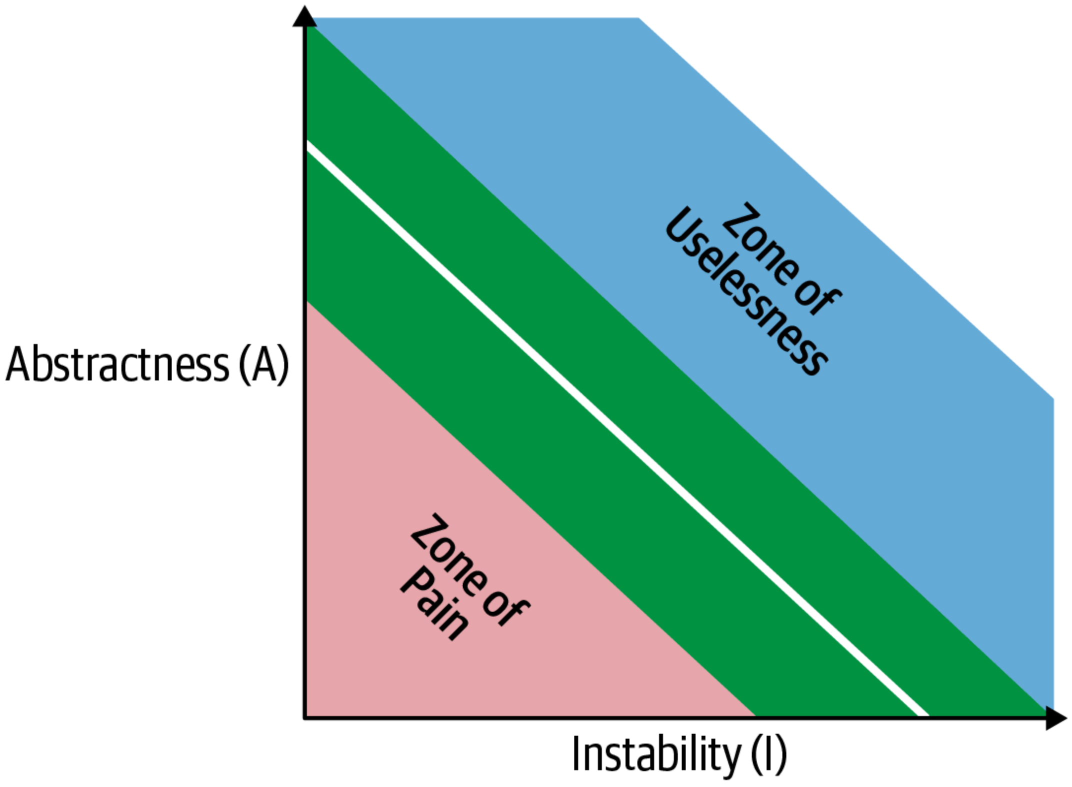

Figure 3-4. The Zones of Uselessness and Pain

Many platforms provide tools to calculate these measures, which assist architects when analyzing code bases to get familiar with them, prepare for a migration, or assess technical debt.

The Limitations of Metrics

While the industry has a few code-level metrics that provide valuable insight, our tools are extremely blunt compared to analysis tools in other engineering disciplines. Even metrics derived directly from the structure of code require interpretation. For example, Cyclomatic Complexity (see “Cyclomatic Complexity”) measures complexity in code bases, but this metric cannot distinguish between *essential* complexity (the code is complex because the underlying problem is complex) and *accidental complexity* (the code is more complex than it should be). Virtually all code-level metrics require interpretation, but it is still useful to establish baselines for critical metrics such as Cyclomatic Complexity so that architects can assess which type the code base exhibits. We discuss setting up just such tests in “Governance and Fitness Functions”.

Yourdon and Constantine’s *Structured Design*, published in 1979, predates the popularity of object-oriented languages. It focuses instead on structured programming constructs, such as functions (not methods). It also defines other types of coupling that are outdated because of modern program language design. Object-oriented programming introduced additional concepts that overlay afferent and efferent coupling, including a more refined vocabulary to describe coupling called *connascence*.

Connascence

Meilir Page-Jones’s book *What Every Programmer Should Know about Object-Oriented Design* (Dorset House, 1996) created a more precise *language* to describe different types of coupling in object-oriented languages. Connascence isn't a coupling metric like afferent and efferent coupling—rather, it represents a language that helps architects describe different types of coupling more precisely (and understand some common consequences of types of coupling).

Two components are *connascent* if a change in one would require the other to be modified in order to maintain the overall correctness of the system. Page-Jones distinguishes two types of connascence: *static* and *dynamic*.

Static connascence

Static connascence refers to source-code-level coupling (as opposed to execution-time coupling, covered in "Dynamic connascence"). Architects view static connascence as the *degree* to which something is coupled via either afferent or efferent coupling. There are several types of static connascence:

Connascence of Name

> Multiple components must agree on the name of an entity.
>
> Method names and method parameters are the most common way that code bases are coupled and the most desirable, especially in light of modern refactoring tools that make system-wide name changes trivial to implement. For example, developers no longer change the name of a method in an active code base but rather *refactor* the method name using modern tools, affecting the change throughout the code base.

Connascence of Type

> Multiple components must agree on the type of an entity.
>
> This type of connascence refers to the common tendency in many statically typed languages to limit variables and parameters to specific types. However, this capability isn't purely for statically typed languages—some dynamically typed languages also offer selective typing, notably Clojure and Clojure Spec.

Connascence of Meaning

> Multiple components must agree on the meaning of particular values. It is also called *Connascence of Convention*.
>
> The most common obvious case for this type of connascence in code bases is hardcoded numbers rather than constants. For example, it is common in some languages to consider defining somewhere that `int TRUE = 1; int FALSE = 0`. Imagine the problems that would arise if someone flipped those values.

Connascence of Position

> Multiple components must agree on the order of values.
>
> This is an issue with parameter values for method and function calls, even in languages that feature static typing. For example, if a developer creates a method `void updateSeat(String name, String seatLocation)` and calls it with the values `updateSeat("14D", "Ford, N")`, the semantics aren't correct, even if the types are.

Connascence of Algorithm

Multiple components must agree on a particular algorithm.

A common case for *Connascence of Algorithm* occurs when a developer defines a security hashing algorithm that must run and produce identical results on both the server and client to authenticate the user. Obviously, this represents a high degree of coupling—if any details of either algorithm change, the handshake will no longer work.

Dynamic connascence

The other type of connascence Page-Jones defines is *dynamic connascence*, which analyzes calls at runtime. The types of dynamic connascence include:

Connascence of Execution

The order of execution of multiple components is important.

Consider this code:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

It won't work correctly because certain properties must be set in a specific order.

Connascence of Timing

The timing of the execution of multiple components is important.

The common case for this type of connascence is a race condition caused by two threads executing at the same time, affecting the outcome of the joint operation.

Connascence of Values

Several values depend on one another and must change together.

Consider a case where a developer has defined a rectangle by defining four points to represent its corners. To maintain the integrity of the data structure, the developer cannot randomly change one of the points without considering the impact on the other points to preserve the shape of the rectangle.

A more common and problematic case involves transactions, especially in distributed systems. In a system designed with separate databases, when someone needs to update a single value across all of the databases, the values must either all change together or not change at all.

Connascence of Identity

Multiple components must reference the same entity.

A common example of *Connascence of Identity* involves two independent components that must share and update a common data structure, such as a distributed queue.

Connascence properties

Connascence is an analysis framework for architects and developers, and some of its properties help ensure that we use it wisely. These connascence properties include:

Strength

> Architects determine the *strength* of a system's connascence by the ease with which a developer can refactor its coupling. Some types of connascence are demonstrably more desirable than others, as shown in Figure 3-5. Refactoring toward better types of connascence can improve the coupling characteristics of a code base.
>
> Architects should prefer static connascence to dynamic because developers can determine it by simple source-code analysis, and because modern tools make it trivial to improve static connascence. For example, *Connascence of Meaning* could be improved by refactoring to *Connascence of Name*, creating a named constant rather than a magic value.

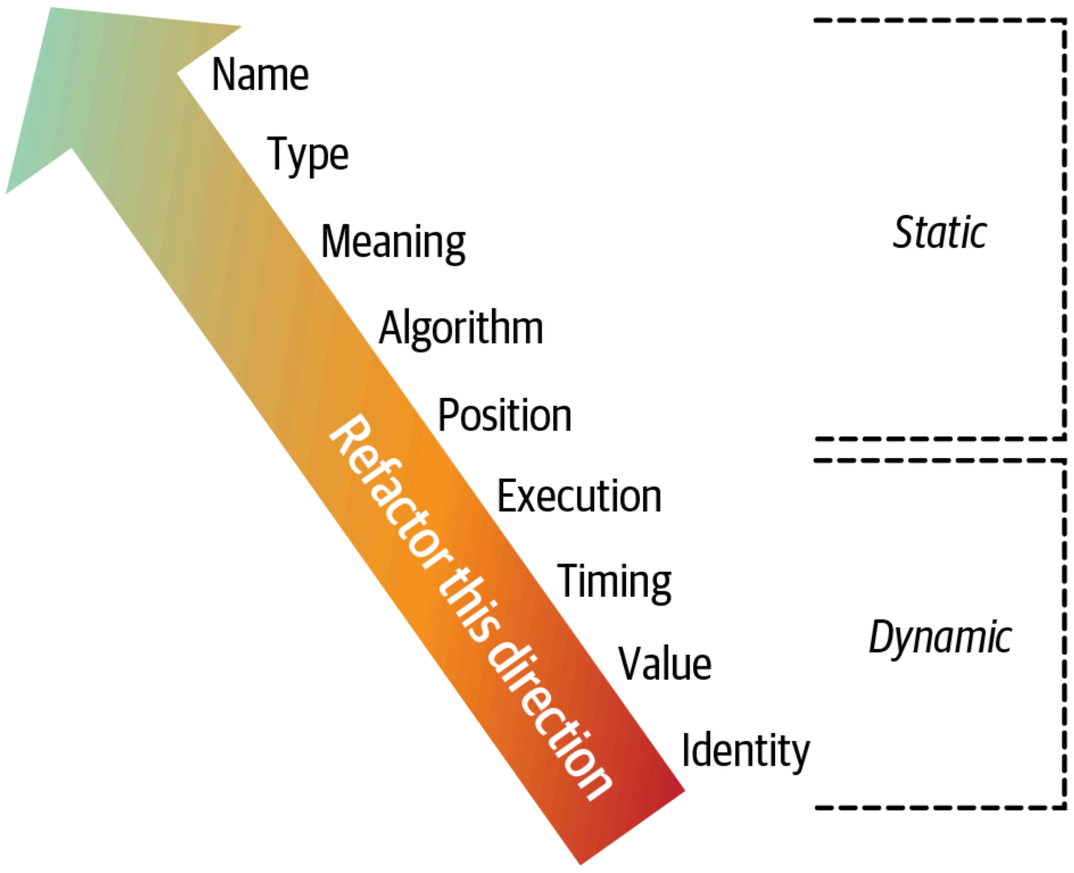

Figure 3-5. Connascence *strength* can be a good refactoring guide

Locality

> The *locality* of a system's connascence measures how proximal (close) its modules are to each other in the code base. *Proximal code* (code in the same module) typically has more and higher forms of connascence than more separated code (in separate modules or code bases). In other words, forms of connascence that would indicate poor coupling when the components are far apart are fine when the components are closer together. For example, if two classes in the same module have *Connascence of Meaning*, it is less damaging to the code base than if those classes were in different modules.

> Architects were largely unaware of the importance of this observation when the author first published it. In modern terms, he is suggesting that architects should limit the scope of implementation details (high coupling) as narrowly as practical, which is the same advice derived from domain-driven design's (DDD) bounded context idea. The architectural observation is the same—limit implementation coupling. Meilir-Page described a good design principle that was reintroduced more fully via DDD (see "Domain-Driven Design's Bounded Context" in Chapter 7).
>
> It's a good idea to consider strength and locality together. Stronger forms of connascence within the same module represent less "code smell" than the same connascence spread apart.

Degree

> The *degree* of connascence relates to the size of the impact of changing a class in a particular module—does that change impact a few classes or many? Lesser degrees of connascence require fewer changes to other classes and modules and hence damage code bases less. In other words, having high dynamic connascence isn't terrible if an architect only has a few modules. However, code bases tend to grow, making a small problem correspondingly bigger in terms of change.

In *What Every Programmer Should Know about Object-Oriented Design*, Page-Jones offers three guidelines for using connascence to improve system modularity:

- Minimize overall connascence by breaking the system into encapsulated elements.
- Minimize any remaining connascence that crosses encapsulation boundaries.
- Maximize connascence within encapsulation boundaries.

The legendary software architecture innovator Jim Weirich, who repopularized the concept of connascence, offers two great rules from his talk on "Connescence Examined" during the Emerging Technologies for the Enterprise conference in 2012:

> Rule of Degree: Convert strong forms of connascence into weaker forms of connascence.
>
> Rule of Locality: As the distance between software elements increases, use weaker forms of connascence.

Architects benefit from learning about connascence for the same reason it's beneficial to learn about design patterns: connascence provides more precise language to describe different types of coupling. For example, an architect can tell someone, "We need a service and there can only be one instance," or they can tell someone, "We need a Singleton service." The Singleton design pattern nicely encapsulates the context and solution to a common problem with a simple name.

Similarly, when performing a code review, an architect can instruct a developer, "Don't add a magic string constant in the middle of a method declaration. Extract it as a constant instead." Or they could say, "You have *Connascence of Meaning*; refactor it to *Connascence of Name*."

From Modules to Components

We use the term *module* throughout this book as a generic name for bundles of related code. However, most architects refer to modules as *components*, the key building blocks for software architecture. This concept of a *component*, and the corresponding analysis of logical or physical separation, has existed since the earliest days of computer science, yet developers and architects still struggle to achieve good outcomes.

We'll discuss deriving components from problem domains in Chapter 8, but we must first discuss another fundamental aspect of software architecture: architectural characteristics and their scope.