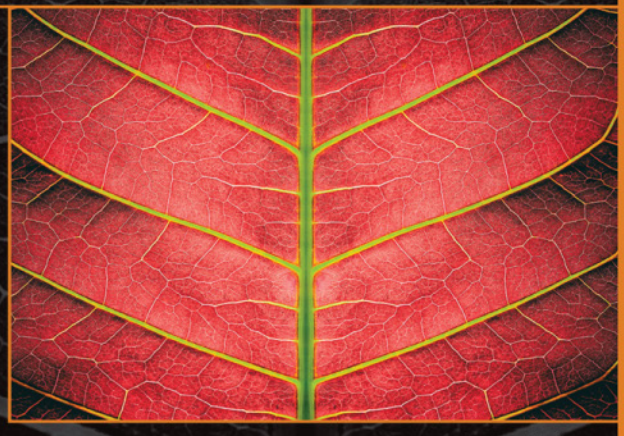
The Addison Wesley Signature Series

A Vaughn Vernon Signature Book

Balancing Coupling in Software Design

Universal Design Principles for Architecting Modular Software Systems

Vlad Khononov



Forewords by
Rebecca Wirfs-Brock
and Kent Beck

Praise for *Balancing Coupling in Software Design*

"Your software can get easier to change over time, but it's hard work to make that happen. With the concepts and skills you'll gain from this book, though, you will be well on your way."

—Kent Beck, Creator of Extreme Programming and Test-Driven Development

"Coupling is one of those words that is used a lot, but little understood. Vlad propels us from simplistic slogans like 'always decouple components' to a nuanced discussion of coupling in the context of complexity and software evolution. If you build modern software, read this book!"

—Gregor Hohpe, author of The Software Architect Elevator

"Get ready to unravel the multi-dimensional nature of coupling and the forces at work behind the scenes. The reference for those looking for a means to both assess and understand the real impact of design decisions."

—Chris Bradford, Director of Digital Services, Cambridge Consultants

"Coupling is a tale as old as software. It's a difficult concept to grasp and explain, but Vlad effortlessly lays out the many facets of coupling in this book, presenting a tangible model to measure and balance coupling in modern distributed systems. This is a must-read for every software professional!"

—Laila Bougria, solutions architect & engineer

"This book is essential for every software architect and developer, offering an unparalleled, thorough, and directly applicable exploration of the concept of coupling. Vlad's work is a crucial resource that will be heavily quoted and referenced in future discussions and publications."

—Michael Plöd, fellow @ INNOQ

"Every software engineer is sensitive to coupling, the measure of interconnection between parts. Still, many times the understanding of such a fundamental property remains unarticulated. In this book, Vlad introduces a much-needed intellectual tool to reason about coupling in a systematic way, offering a novel perspective on this essential topic."

—Ilio Catallo, senior software engineer

"Coupling is among the most slippery topics in software development. However, with this book, Vlad simplifies for us how coupling, from a great villain, can become a design tool when well understood. This is an indispensable guide for anyone dealing with software design—especially complex ones."

—William Santos, software architect

"*Balancing Coupling in Software Design* by Vlad Khononov is an essential read for architects aiming for quality, evolvable systems. Khononov expertly classifies dependencies and reveals how varying designs impact effort based on component distance and change frequency, introducing a unified metric for coupling. With insightful case studies, he guides readers toward achieving optimal modularity and long-term system adaptability by illustrating and rectifying imbalances."

—Asher Sterkin, independent software technology expert

"Khononov's groundbreaking work unifies paramount forces of software design into a coherent model for evaluating coupling of software systems. His insights provide an invaluable framework for architects to design modular, evolving systems that span legacy and modern architectures."

—Felipe Henrique Gross Windmoller, staff software engineer, Banco do Brasil

"This book systematizes over five decades of software design knowledge, offering a comprehensive guide on coupling, its dimensions, and how to manage it effectively. If software design is a constant battle with complexity, then this book is about mastering the art of winning."

—Ivan Zakervsky, IT architect

To reiterate, a module is a boundary encompassing a well-defined functionality, which it exposes for use by other parts of the system. Consequently, a module could represent nearly any type of logical or physical boundary within a software system, be it a service, a namespace, an object, or something else.

Throughout this book, I'll use the term "module" to signify a boundary enclosing specific functionality. This functionality is exposed to external consumers and either is or has the potential to be independently compiled.

Function, Logic, and Context of Software Modules

We can use the three properties of a module—function, logic, and context—to describe all kinds of the aforementioned software modules.

Function

A software module's function is the functionality it exposes to its consumers over its public interface. For example:

- A service's functionality can be exposed through a REST API or asynchronously through publishing and subscribing to messages.
- An object's function is expressed in its public methods and members.
- The function of a namespace, package, or distributed library consists of the functionality implemented by its members.
- If a distinct method or a function is treated as a module, its name and signature reflect its function.

Logic

A software module's logic encompasses all the implementation and design decisions that are needed to implement its function. It includes its source code,[4] as well as internal infrastructural components (e.g., databases, message buses) that are not needed for describing the module's function.

4. Rumor has it that this is where the term "business logic" comes from. This implies that there are different kinds of "logics" encompassed in a module: logic for integrating infrastructural components, and logic for business tasks. That said, I couldn't find any sources that can prove this observation.

Context

All types of software modules depend on various attributes of their execution environments and/or make assumptions regarding the context in which they operate. For example:

- At a very basic level, a certain runtime environment is needed to execute a module. Moreover, a specific version of the runtime environment may be required.
- A certain level of compute resources, such as CPU, memory, or network bandwidth, may be needed for the module to function properly.
- A module may assume that the calls are pre-authorized instead of performing authorization itself.

Going back to the definition of a module's context, the main difference between function and context is that the assumptions and requirements tied to the context are not reflected in the module's public interface—its function.

Now that you have a solid understanding of what a software module is, let's delve into the design considerations for designing a modular system.

Effective Modules

As noted in the previous sections, an arbitrary decomposition of a system into components won't make it modular. The hierarchical nature of modules doesn't make it any easier. Failing to properly design modules at any level in the hierarchy can potentially undermine the whole effort.

Effective design of modules is not trivial, and failures to do so can be spotted all across the history of software engineering. For example, not so long ago, many believed that a microservices-based architecture is the easy solution for designing flexible, evolvable systems. However, without a proper principle guiding the decomposition of a system into microservices (modules), many teams ended up with distributed monoliths—solutions that were much less flexible than the original design. As they say, history tends to repeat itself, and almost exactly the same situation happened when modularity was introduced to software design:

> When I came on the scene (in the late 1960s) software development managers had realized that building what they called monolithic systems wasn't working. They wanted to divide the work to be done into parts (which they called modules) and each part or module would be assigned to a different team or team member. Their hope was that (a) when they put the parts together they would "fit" and the system would work and (b) when they had to make changes, the changes would be confined to a single module. Neither of those things happened. The reason was that they were doing a "bad job"

> of dividing the work into modules. Those modules had very complex interfaces, and changes almost always affected many modules. —David L. Parnas, personal correspondence to author (May 3, 2023)

Following that experience, Parnas (1971) proposed a principle intended to guide more effective decomposition of systems into modules: information hiding. According to the principle, an effective module is one that hides decisions. If a decision has to be revisited, the change should only affect one module, the one that "hides" it, thus minimizing cascading changes rippling across multiple components of the system.

In Parnas's later work (1985, 2003), he equated modules following the information-hiding principle to the concept of abstraction. Let's see what an abstraction is, what makes an effective abstraction, and how to use this knowledge to craft module boundaries.

Modules as Abstractions

The goal of an abstraction is to represent multiple things equally well. For example, the word "car" is an abstraction. When thinking about a "car," one does not need to consider a specific make, model, or color. It could be a Tesla Model 3, an SUV, a taxi, or even a Formula 1 race car; it could be red, blue, or silver. These specific details are not necessary to understand the basic concept of a car.

For an abstraction "to work," it has to eliminate details that are relevant to concrete cases but are not shared by all. Instead, to represent multiple things equally well, it has to focus on aspects shared by all members of a group. Going back to the previous example, the word "car" simplifies our understanding by focusing on the common characteristics of all cars, such as their function of providing transportation and their typical structure, which often includes four wheels, an engine, and a steering wheel.

By focusing only on the details that are shared by a group of entities, an abstraction hides decisions that are likely to change. As a result, the more general an abstraction is, the more stable it is. Or, the fewer details that are shared by an abstraction, the less likely it is to change.

> **Note**
>
> Interestingly, the term "software module" is an abstraction itself. As you learned in the preceding section, a software module can represent a variety of boundaries, including services, namespaces, and objects. That's the concept of abstraction in action. It eliminates details relevant to concrete types of software boundaries, while focusing on what is essential: responsibility assignment, or the encapsulated functionality. Hence, you can use the term "module" to represent all kinds of software boundaries equally well.

A well-designed module is an abstraction. Its public interface should focus on the functionality provided by the module, while hiding all the details that are not shared by all possible implementations of that functionality. Going back to the example of a repository object in Chapter 3, the interface described in Listing 4.1 focuses on the required functionality, while encapsulating the concrete implementation details.

Listing 4.1 *A Module Interface That Focuses on the Functionality It Provides, While Encapsulating Its Implementation Details*

```
interface CustomerRepository {
    Customer Load(CustomerId id);
    void Save(Customer customer);
    Collection<Customer> FindByName(Name name);
    Collection<Customer> FindByPhone(PhoneNumber phone);
}
```

A concrete implementation of the repository could use a relational database, a document store, or even a polyglot persistence–based implementation that leverages multiple databases. Moreover, this design allows the consumers of the repository to switch from one concrete implementation to another, without being affected by the change.

The notion of effortlessly switching from one database to another often has a somewhat questionable reputation within the software engineering community. Such changes aren't common.[5] That said, there's a more frequent and crucial need to switch the implementation behind a stable interface. When you're altering a module's implementation without changing its interface, such as fixing a bug or changing its behavior, you're essentially replacing its implementation. For example, the kinds of queries used in the `FindByName()` and `FindByPhone()` methods can be changed even when retaining the use of the same database. It could be that an index, name, and phone number are added to the database schema itself. Or it could be that the data is restructured to better optimize queries. Neither of these changes should impact the client's use of the module interface.

That said, the possibility of switching an implementation is not the only goal of introducing an abstraction. As Edsger W. Dijkstra (1972) famously put it, "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

It may seem that using an abstraction introduces vagueness or lack of detail. However, as Dijkstra argues, that's not the goal. Instead, an abstraction should create a new level of understanding—a "semantic level"—where one can be "absolutely precise." Balance is needed to reach a proper level of abstraction to convey the correct semantics. Consider this: If you use an abstraction called "vehicle" to represent

5. With the exception of running a suite of unit tests that replace a physical database with an in-memory mock.

cars, it might be an overly broad generalization. Ask yourself: Are you actually modeling a range of vehicles, such as motorcycles and buses, necessitating such a wide-ranging abstraction? If the answer is no, then using "car" as your abstraction is more appropriate and precisely conveys the intended meaning.

By focusing on the essentials—functionality of modules—while ignoring extraneous information, abstractions allow us to reason about complex systems without getting lost in the details. A common example of a modular system is a personal computer. We can reason about the interactions of its modules—CPU, motherboard, random-access memory, hard drive, and others—all without understanding the intricate technicalities of each individual component. When troubleshooting a problem, we don't need to comprehend how a CPU processes instructions or how a hard drive stores data at a microscopic level. Instead, we consider their roles within the larger system: a new semantic level provided by effective abstractions.

Finally, abstractions, like modules, are hierarchical. In software design, "levels of abstraction"[6] are used to refer to different levels of detail when reasoning about systems. Higher levels of abstraction are closer to user-facing functionality, while lower levels are more about components related to low-level implementation details. Different levels of detail require different languages for discussing the functionalities implemented at each level. Those languages, or (as Dijkstra called them) semantic levels, are formed by designing abstractions.

Hierarchical abstractions also serve as further illustration of modularity's hierarchical nature. Since abstractions adhere to the same design principles at all levels, modular design exhibits not only a hierarchical but also a fractal structure. In upcoming chapters, I will discuss in detail how the same rules govern modular structures at different scales. But for now, let's revisit the topic of the previous chapters, complexity, and analyze its relationship with modularity.

Modularity, Complexity, and Coupling

Poor design of a system's modules leads to complexity. As we discussed in Chapter 3, complexity can be both local and global, while the exact meaning of local/global depends on point of view: Global complexity is local complexity at a higher level of abstraction, and vice versa. But what exactly makes one design modular and another one complex?

Both modularity and complexity result from how knowledge is shared across the system's design. Sharing extraneous knowledge across components increases the

6. Or layers of abstraction.

cognitive load required to understand the system and introduces complex interactions (unintended results, or intended results but in unintended ways).

Modularity, on the other hand, controls complexity of a system in two ways:

1. Eliminating accidental complexity; in other words, avoiding complexity driven by the poor design of a system.
2. Managing the system's essential complexity. The essential complexity is an inherent part of the system's business domain and, thus, cannot be eliminated. On the other hand, modular design contains its effect by encapsulating the complex parts in proper modules, preventing its complexity from "spilling" across the system.

In terms of knowledge, modular design optimizes how knowledge is distributed across the components (modules) of a system.

Essentially, a module is a knowledge boundary. A module's boundary defines what knowledge will be exposed to other parts of the system and what knowledge will be encapsulated (hidden) by the module. The three properties of a module that were introduced earlier in the chapter define three kinds of knowledge reflected by the design of a module:

1. Function: The explicitly exposed knowledge
2. Logic: Knowledge that is hidden within the module
3. Context: Knowledge the module has about its environment

An effective design of a module maximizes the knowledge it encapsulates, while sharing only the minimum that is required for other components to work with the module.

Deep Modules

In his book *A Philosophy of Software Design*, John Ousterhout (2018) proposes a visual heuristic for evaluating a module's boundary. Imagine that a module's function and logic are represented by a rectangle, as illustrated in Figure 4.3. The rectangle's area reflects the module's implementation details (logic), while the bottom edge is the module's function (public interface).