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

FREE SAMPLE CHAPTER

Praise for *Balancing Coupling in Software Design*

"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* is a must-read for any software architect. Vlad Khononov masterfully demystifies coupling, offering practical insights and strategies to balance it effectively. This book is invaluable for creating modular, scalable, and maintainable software systems. Highly recommended!"

—*Vadim Solovey, CEO at DoiT International*

"*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*

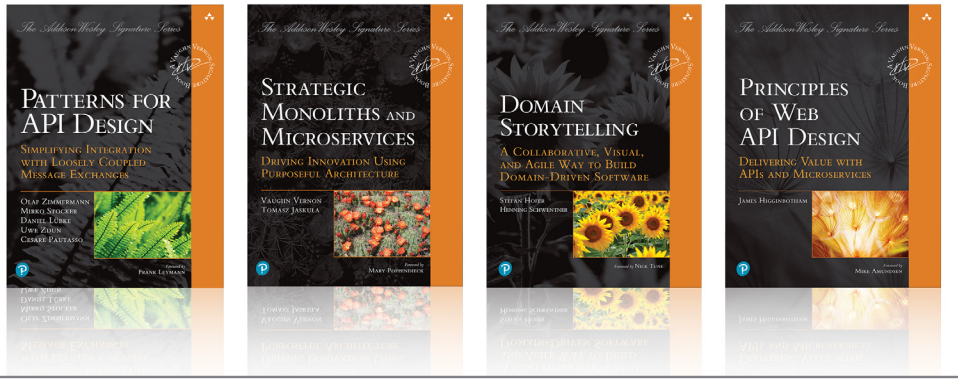
"Khononov's groundbreaking work unifies paramount forces of software design into a coherent model for evaluating coupling of software systems. His insights provide an invaluable framework for architects to design modular, evolving systems that span legacy and modern architectures."

—*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*

Pearson Addison-Wesley Signature Series



Visit informit.com/awss/vernon for a complete list of available publications.

The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Vaughn Vernon is a champion of simplifying software architecture and development, with an emphasis on reactive methods. He has a unique ability to teach and lead with Domain-Driven Design using lightweight tools to unveil unimagined value. He helps organizations achieve competitive advantages using enduring tools such as architectures, patterns, and approaches, and through partnerships between business stakeholders and software developers.

Vaughn's Signature Series guides readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

Balancing Coupling in Software Design

Universal Design Principles for Architecting Modular Software Systems

Vlad Khononov

Addison-Wesley

Hoboken, New Jersey

Cover image: pernsanitfoto/Shutterstock

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024942574

ISBN-13: 978-0-13-735348-4
ISBN-10: 0-13-735348-0

$PrintCode

Dedicated to everyone who kept asking when this book would finally be published.

#AdoptDontShop

This page intentionally left blank

Contents

This page intentionally left blank

Series Editor Foreword

I recall meeting Vladik at a conference or two nearly a decade ago, by the publication date of this, his new book. I recall Vladik, or Vlad if you like, being a quiet and thoughtful person, and with a good sense of humor, which scored high with me. He's not overly quiet though, as he's proven by his insightful conference talks. Since that time, we met up now and then, with our last in-person opportunity in New York City at a software architecture conference just prior to the COVID-19 lockdown. Although I find that reference point distasteful, it was thereafter a pivotal time when my signature series got life. Shortly thereafter, I asked Vladik if he would write a book for it. To my delight, he agreed. During the following years, Vladik encountered several challenges, some of a personal family nature, and others dealing with life and work during the crazy pandemic period. Yet, he endured and persisted in his work. I reviewed Vladik's book a few different times and watched it transition from rough draft to finished product. I have to say that experiencing the blend of past practices framed in a fresh and powerful way was fascinating. I'll explain more about that after I introduce the purpose of this series.

My Signature Series is designed and curated to guide readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, as well as functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

From here I am focusing now on only two words: organic refinement.

The first word, *organic*, stood out to me recently when a friend and colleague used it to describe software architecture. I have heard and used the word *organic* in connection with software development, but I didn't think about that word as carefully as I did then when I personally consumed the two used together: *organic architecture*.

Think about the word *organic*, and even the word *organism*. For the most part these are used when referring to living things, but are also used to describe inanimate things that feature some characteristics that resemble life forms. *Organic* originates in Greek. Its etymology is with reference to a functioning organ of the body. If you read the etymology of *organ*, it has a broader use, and in fact organic followed suit: body organs; to implement; describes a tool for making or doing; a musical instrument.

We can readily think of numerous organic objects—living organisms—from the very large to the microscopic single-celled life forms. With the second use of organism, though, examples may not as readily pop into our mind. One example is an organization, which includes the prefix of both *organic* and *organism*. In this use of *organism*, I'm describing something that is structured with bidirectional dependencies. An organization is an organism because it has organized parts. This kind of organism cannot survive without the parts, and the parts cannot survive without the organism.

Taking that perspective, we can continue applying this thinking to nonliving things because they exhibit characteristics of living organisms. Consider the atom. Every single atom is a system unto itself, and all living things are composed of atoms. Yet, atoms are inorganic and do not reproduce. Even so, it's not difficult to think of atoms as living things in the sense that they are endlessly moving, functioning. Atoms even bond with other atoms. When this occurs, each atom is not only a single system unto itself but also becomes a subsystem along with other atoms as subsystems, with their combined behaviors yielding a greater whole system.

So then, all kinds of concepts regarding software are quite organic in that nonliving things are still "characterized" by aspects of living organisms. When we discuss software model concepts using concrete scenarios, or draw an architecture diagram, or write a unit test and its corresponding domain model unit, software starts to come alive. It isn't static, because we continue to discuss how to make it better, subjecting it to refinement, where one scenario leads to another, and that has an impact on the architecture and the domain model. As we continue to iterate, the increasing value in refinements leads to incremental growth of the organism. As time progresses so does the software. We wrangle with and tackle complexity through useful abstractions, and the software grows and changes shapes, all with the explicit purpose of making work better for real living organisms at global scales.

Sadly, software organics tend to grow poorly more often than they grow well. Even if they start out life in good health, they tend to get diseases, become deformed, grow unnatural appendages, atrophy, and deteriorate. Worse still is that these symptoms are caused by efforts to refine the software that go wrong instead of making things better. The worst part is that with every failed refinement, everything that goes wrong with these complexly ill bodies doesn't cause their death. Oh, if they could just die! Instead, we have to kill them and killing them requires nerves, skills, and the intestinal fortitude of a dragon slayer. No, not one, but dozens of vigorous dragon slayers. Actually, make that dozens of dragon slayers who have really big brains.

That's where this series comes into play. I am curating a series designed to help you mature and reach greater success with a variety of approaches—reactive, object,

and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs. And along with that, the series covers best uses of the associated underlying technologies. It's not accomplished at one fell swoop. It requires organic refinement with purpose and skill. I and the other authors are here to help. To that end, we've delivered our very best to achieve our goal.

Is balancing software coupling organic? Absolutely! Start with a sinkhole of a repository where all goopy code has gone to collect as sludge. Triple yuk! How can you possibly grow any new, bright life from the quagmire? Simple. Start scooping and separating, add some good soil and nutrients, build some modular containers around the mounds of enriched earth, and start planting seeds in each—some common, some special, and even a few exotics. Before you know it, poof, and there's fresh life!

Well, sort of, but not exactly. You'll have to learn about "software gardening." That includes soaking up the basic almanac of coupling: what coupling is exactly; the bad and the good of it; how coupling relates to system design and levels of complexity; and how modularity helps, of course. After you are on solid ground, there's a whole set of dimensions to learn that will help you evaluate the environment for sustained growth: strength, space, and time. There's the introduction to module coupling and connascence, which leads to Vladik's own new model: integration strength. This might flow like flood irrigation but keep gulping. What about distance and how it plays into different crops being planted and nourished, and how can cultivating and pruning one crop lead to positive and [or?] negative impacts on another? It's sprouting.

"Wait a minute, let me catch up," you say? That's a fitting response to how time plays into planting rotations and potential volatility due to various elements. All this requires balance to avoid the enemy of all software; that is, *growth over time*. Examples of how other gardens have grown will help your plantings to sustain life despite those harsh elements. And it works because it's all backed by decades of research and development by renown software practitioners—umm, horticulturalists.

You are now ready to roll up your sleeves, open the spigot, and absorb. Get to growing excellent software!

—*Vaughn Vernon*

This page intentionally left blank

Foreword

Successful software systems grow and evolve—to add new features and capabilities, and to support new technologies and platforms. But is it inevitable that over time they become unmaintainable "Big Balls of Mud?"

Well, given that complex software systems are structured out of modular interrelated units of functionality, each with discrete responsibilities, there will always be coupling. But the ways modules communicate and how they share information have implications on our ability to change them.

In this book, Vlad, after informing us of the original design ideas of module coupling and connascence, updates us with fresh ways to think about the various dimensions of coupling: integration strength, volatility, and distance. He then leads us on a journey to deeply understanding coupling and offers a comprehensive model for evaluating coupling options when designing or refactoring parts of a system. Most authors explain coupling in a paragraph or a page. Vlad gives us an entire book.

There are various ways we can reduce coupling, and Vlad explains them. Should we always look at coupling as something bad? Of course not. But we shouldn't be complacent either. The last section of Vlad's book introduces the notion of balanced coupling, and a process for thinking through design implications as you "rebalance" the coupling in your system.

Thanks, Vlad, for persisting in writing this comprehensive treatment of coupling, balancing, and then rebalancing (design always involves trade-offs). You provide us with a wealth of new and insightful ways to think about structuring and restructuring complex systems to keep them working and evolvable. This book gives me hope that in the hands of thoughtful designers, software system entropy isn't inevitable.

—Rebecca J. Wirfs-Brock
May 2, 2024

This page intentionally left blank

Foreword

Design happens in the cracks.

At first as a programmer you don't even know what the *things* are. You learn about functions. You learn about types. You learn about classes and modules and packages and services. You still haven't learned to design. You can make all the things, but you can't design. Because design happens in the cracks.

Design prepares change. The things, those are the change. Design makes places for the new things, the functions and types and classes and modules and packages and services.

What Vlad has done is catalog the cracks, the seams, the dark squishy in between of software. If you want to not just make changes, but make changes easy, this is the vocabulary you'll need. The glossary. The dictionary of cracks.

Vlad understands well that experts learn by doing and reflecting. The review questions with each chapter are stepping stones to learning for those willing to put in the work required to learn.

Since Vlad did me the honor of inviting me to invite you to read this book, I'll take a moment to complain about vocabulary. Vlad uses "integration strength" to mean what I mean by "coupling", the relationship between elements where changing one in a particular way requires changing the other. He uses "coupling" to mean a more general connection between elements, at runtime or compile time. It's not a huge deal but it's important for me to say.

Having said that, I heartily recommend reading *Balancing Coupling in Software Design*. Strike that. I heartily recommend *learning* from *Balancing Coupling in Software Design*. Read about a kind of crack, a connection, go find it in your own code, find it in other people's code, try out variations on it, try out timings for changing it, watch how it affects the behavioral changes you want to make. Then read about another kind of crack. Compare and contrast. Dig in.

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
San Francisco, California
2024

This page intentionally left blank

Preface

Books on software design typically dedicate a few pages to coupling. On rare occasions, you'll find a whole chapter on the subject. Yet, while fads come and go, coupling has been, is, and, I bet, always will be relevant. Don't believe me? Just take a moment and listen to the industry chatter. You will hear the "coupling is bad" mantra everywhere. But what exactly is this "coupling" thing? Is it always that bad, or does it become really bad after a certain point? Can you even measure it? If so, how? These are the questions I've sought answers to since I started working as a software engineer. All I encountered was more and more of "Avoid coupling!" or "This architectural pattern will save you from coupling!" or, even worse, "The only way to avoid coupling is to use our product!" Sigh.

Around 2014, yet another "decoupling salvation" emerged: microservices. I even remember a slide from some conference that read "Microservices is the architecture for decoupling." It was "microservices this" and "microservices that," but back then nobody could really define what a microservice was. That didn't stop me (or anyone else) from trying. Pumped by the microservices/decoupling hype, we aimed to "decouple" everything in the project I was working on. For that, we designed microservices around business entities, with each API resembling mostly CRUD[1] operations. Each entity can be evolved independently, we said. The result? A fiasco. No, a cosmic-scale fiasco.

That failed project, however, turned out to be a blessing in disguise. I had to figure out why what promised decoupling resulted in a coupling Godzilla. I had to get it right. So, I set out to read all the papers and books that could explain how to do microservices better. Eventually, I found an explanation. All our design mistakes were described in Chapter 6 of the book *Structured Design* (Yourdon and Constantine 1975). The title of the chapter? "Coupling."

That's how my journey into coupling in software design started. I wanted to learn everything that we knew but had forgotten. A few years later, all the puzzle pieces started falling into place. Everything I learned began to form a coherent picture—a three-dimensional model of how coupling affects software projects. Gradually, I started applying this model in my day-to-day work. It worked! What's more, it completely changed the way I think about software design.

1. Create, Read, Update, and Delete.

At some point, I couldn't keep it inside anymore, and I wanted to share my findings. This led to a talk I gave at the Domain-Driven Design Europe 2020 conference, titled "Balancing Coupling in Distributed Systems." As I was walking off the stage, cortisol and adrenaline were conducting a stress hormone conference of their own in my bloodstream. The only thing I remember is Rebecca Wirfs-Brock telling me that I had to keep developing these ideas and to write a book about it. Who am I to argue with Rebecca Wirfs-Brock?

I had to write this book for the same reason I gave that conference talk. All this knowledge that we have, but have forgotten, is far too important. So, if you're reading these words, it means that after long years of hard work, I managed to finish this book before it could finish me. I wholeheartedly believe that this material will be as useful to you as it has been to me.

Who Should Read This Book?

As I'm writing this Preface, Pearson's style guidelines instruct me to "be precise and resist the temptation to create a long list of potential readers." Well, then, I will define the book's target audience as people who *create* software.

Whether you are a junior, senior, or principal software engineer or architect, as long as you are making software design decisions at any level of abstraction, coupling can make or break your efforts. Learning to tame the forces of coupling is essential for building modular and evolvable systems.

How This Book Is Organized

This book is divided into three parts.

Part I, Coupling—The first part of the book is about the big picture: how coupling fits in the contexts of software design, complexity, and modularity.

Chapter 1, Coupling and System Design—In the first part of this chapter, you will learn what systems are, how they are built, and the role coupling plays in any system. The second part of the chapter switches the focus to software systems and introduces the terminology that will be used to describe coupling in the chapters that follow.

Chapter 2, Coupling and Complexity: Cynefin—Since complexity is something we would rather avoid, it's important to understand what it is in the first place. To that end, the chapter introduces the basic principles of the Cynefin framework that precisely defines what complexity is.

Chapter 3, Coupling and Complexity: Interactions—This chapter shifts the discussion to systems in general and software design complexity in particular. You will learn what makes a software system complex and what that has to do with coupling.

Chapter 4, Coupling and Modularity—This chapter switches the focus to what we would rather achieve: modularity. It defines the notions of modularity and software modules. Most importantly, it discusses coupling: the rudder that can steer a system toward either complexity or modularity.

Part II, Dimensions—The second part of the book homes in on coupling. You will learn the different ways coupling affects systems and a number of models for evaluating its effect.

Chapter 5, Structured Design's Module Coupling—This chapter starts the journey through time and introduces the first model of evaluating coupling in software design, a model that was formulated in the late 1960s but is still relevant today.

Chapter 6, Connascence—This chapter introduces a model that reflects a different aspect of coupling: connascence. You will learn what it means for modules to be "born together" and the different magnitudes of this kind of relationship.

Chapter 7, Integration Strength—Here, we combine the aspects of coupling reflected by structured design's module coupling and connascence into a combined model known as integration strength. You will learn to use this model to evaluate the knowledge shared among the components of a system.

Chapter 8, Distance—In this chapter, we switch the focus to a different dimension: space. You will learn how the physical position of modules in a codebase can affect their coupling.

Chapter 9, Volatility—Here, we switch the focus to the dimension of time. We will discuss the reasons for changes in software modules, how a module's volatility can propagate across the system, and how you can evaluate a module's expected rate of change.

Part III, Balance—This part of the book connects the topics in Parts I and II by turning the dimensions of coupling into a tool for designing modular software.

Chapter 10, Balancing Coupling—In this chapter, we explore the insights you can gain by combining the dimensions of coupling. The chapter also introduces the balanced coupling model: a holistic model for evaluating the effects of coupling on the overarching system.

Chapter 11, Rebalancing Coupling—Here, we discuss the strategic evolution of a software system, the changes it brings, and how these changes can be accommodated by rebalancing the coupling forces.

Chapter 12, Fractal Geometry of Software Design—In this chapter, we continue the topic of system evolution, focusing on the most common and important change: growth. This chapter combines knowledge from other industries, and even nature, to uncover the underlying design principles guiding software design.

Chapter 13, Balanced Coupling in Practice—We move from theory to practical application in this chapter by discussing case studies that demonstrate how the balanced coupling model can be used to improve software design. The case studies also demonstrate that the balanced coupling model can be observed at the heart of well-known architectural styles, design patterns, and design principles.

Chapter 14, Conclusion—This chapter summarizes the book's content and provides final advice on applying the learned principles in your day-to-day work.

Case Studies and WolfDesk

This book is grounded in practice: Everything you'll read has been battle-tested and proven useful across multiple software projects and business domains. This real-world experience is reflected in the case studies you'll find in each chapter. Though I can't divulge specific details about the projects, I wanted to provide concrete case studies to make the material less abstract. To do this, I transformed stories from the trenches into case studies about a fictional company, WolfDesk. While the company is fictional, all the case studies are drawn from real projects. Here's a brief description of WolfDesk and its business domain.

WolfDesk

WolfDesk provides a help desk management system as a service. If your startup company needs to offer support to your customers, WolfDesk's solution can get you up and running in no time.

WolfDesk uses a payment model that sets it apart from its competitors. Rather than charging a fee per user, it allows tenants to set up as many users as they need, charging based on the number of support cases opened per billing period. There is no minimum fee, and automatic volume discounts are offered at certain thresholds of monthly cases.

To prevent tenants from exploiting the business model by reusing existing support cases, the lifecycle algorithm ensures that inactive support cases are automatically closed, encouraging customers to open new ones when more support is needed. Furthermore, WolfDesk implements a fraud detection system that analyzes messages and identifies instances of unrelated topics being discussed within the same support case.

In an effort to help tenants streamline their support-related work, WolfDesk has implemented a "support autopilot" feature. This autopilot analyzes new inquiries and attempts to automatically find a matching solution from the tenant's history.

This function helps to further reduce the lifespan of cases, encouraging customers to open new cases for additional questions.

The administration interface allows tenants to configure possible values for support case categories, as well as a list of the tenant's products that require support. To ensure that support cases are routed to agents only during their working hours, WolfDesk allows users to configure different shift schedules for different departments and organizational units.

This page intentionally left blank

Acknowledgments

I would like to extend my deepest gratitude to Vaughn Vernon, without whom this book would not have become a reality. Vaughn not only provided me with the incredible opportunity to bring these ideas to the printed page, but he also supported me throughout the writing process. Thank you for always being there when I needed help, and for your invaluable advice and insights, which have significantly enriched this book.

Haze Humbert is the book's guardian angel or, more formally, its executive editor. The work on this book lasted for four years, and I know I didn't make those years easy for Haze. Everything that could go wrong did, and then some. Haze, you are among the most patient people I know. Thank you for making this happen, and for your support in the moments when I needed it most.

What a relief it was to finish writing this book! However, that was just one battle. To win the war, it needed to be prepared for printing, and what an array of curveballs this entailed. I want to thank Julie Nahil, the book's content producer, for being on my side and helping me get the book laid out and formatted just as I envisioned.

This book transcends different eras of software engineering and diverse fields of study. This is hands down the most challenging project I have ever worked on, and it wouldn't have seen the light of day without the contributions of so many people who helped me along the way. I want to thank the subject matter experts whom I consulted during the writing process[2]: Alistair Cockburn, Gregor Hohpe, Liz Keogh, Ruth Malan, David L. Parnas, Dave Snowden, and Nick Tune.

Heartfelt thanks to the reviewers who were brave enough to read the book's early, unedited drafts, providing feedback that played a crucial role in refining the manuscript: Ilio Catallo, Ruslan Dmytrakovych, Savvas Kleanthous, Hayden Melton, Sonya Natanzon, Artem Shchodro, and Ivan Zakrevsky.

Last but not least, I want to thank two special people from whom I've learned so much, and it's an immense honor to have them as foreword authors: Rebecca J. Wirfs-Brock and Kent Beck. Thank you both for your warm and inspiring words!

2. Whenever I mention a group of people, the list is in alphabetical order by last name.

This page intentionally left blank

About the Author

Vlad (Vladik) Khononov wanted to make his own computer games, so at eight years old, he picked up a book on BASIC. Although he has yet to publish a game, software engineering became his passion and trade. With over two decades of industry experience, Vlad has worked for companies large and small, in roles ranging from webmaster to chief architect. As a consultant and trainer, he currently helps companies make sense of their business domains, untangle legacy systems, and tackle complex architectural challenges.

Vlad maintains an active media career as an author and keynote speaker. Besides the book you are holding, he has written *Learning Domain-Driven Design* (O'Reilly, 2021), which has been translated into eight languages. As a speaker, Vlad has presented at leading software engineering and architecture conferences around the world. He is known for his ability to explain complex concepts in simple, accessible terms, benefitting both technical and nontechnical audiences. You can reach out to Vlad on X (@vladikk) and LinkedIn.

This page intentionally left blank

Chapter 4

Coupling and Modularity

Modularity's perks, we cannot ignore,
But its true essence, we still must explore.
What makes a design coherent and fluent?
It's all about value—future and current.

"95% of the words are spent extolling the benefits of modularity, and little, if anything, is said about how to achieve it" (Myers 1979). These words were written over 40 years ago, but the observation remains true to this day. The significance of modularity is unquestionable: It is the cornerstone of any well-designed system. Yet, despite the many new patterns, architectural styles, and methodologies that have emerged over the years, attaining modularity is still a challenge for many software projects.

The topic of this chapter is modularity and its relationship to coupling. I'll start by defining what modules are and what makes a system modular. Next, you will learn about design considerations that are essential for increasing the modularity of a system and avoiding complex interactions. Ultimately, the chapter discusses the role of cross-component interactions in modular systems, which paves the way for using coupling as a design tool in the chapters that follow.

Modularity

Not only is the notion of modularity not unique to software design, but the term "module" predates software design by about 500 years. At its core, modularity refers to systems composed of self-contained units called modules. At the same time, you

may recall that in Chapter 1, I defined a system as a set of components. This naturally raises an intriguing question: What distinguishes the components of a traditional system from the modules of a modular system?

A system has a goal: the functionality it has to implement. The components of the system are working together to achieve the goal. For example, a social media app enables people to connect, share, and interact, while an accounting system streamlines financial tasks for businesses. However, these functionalities only address the present requirements. As time goes on, the users' needs may change, and new requirements may emerge. That's where modularity comes into play.

Modular design aims to address a wider range of goals than a nonmodular system can. It expands the system's goal to accommodate requirements that are currently unknown but may be needed in the future. Of course, the future requirements are not expected to be available out of the box on day one, but the design should make it possible to evolve the system with a reasonable effort.

By investing in modularity, we design an adaptable and flexible system. That is, the primary goal of modularity is to allow the system to evolve (Cunningham 1992). A famous quote that is often (mis)attributed to Charles Darwin[1] captures this idea perfectly: "It is not the strongest of the species that survives, but the most adaptable." This principle applies to systems as well. Even the most finely tuned, faultlessly performing system of today will face obsolescence if it cannot flex and grow with tomorrow's changes. The less flexible a system is, the less stress it can tolerate. The less stress it can tolerate, the more prone it is to breaking under the pressure of evolving requirements. By being prepared to handle changes, a modular system is better positioned for long-term success.

Modularity also serves as a cognitive tool, streamlining the comprehension of a system. Instead of a monolithic, inscrutable black box, a modular system presents as a collective of individual parts, each performing its function yet able to function collaboratively. This separation into modules allows for a clearer understanding of the system's inner mechanics and how it ultimately delivers the desired output.

But why does this matter? Is it simply a matter of satisfying intellectual curiosity? Not really. A deep understanding of how a system operates is the key to modifying and improving it. This might involve altering existing behavior, such as fixing bugs, or it could involve evolution of the system by introducing new functionalities. The simplicity and transparency of a modular design enable you to tinker, adjust, and innovate more effectively and with more confidence.

With this understanding of the importance of modularity, let's dig deeper into the concept of a module and its role in making a flexible system.

1. Quote Investigator delves into the history of this quotation (https://quoteinvestigator.com/2014/05/04/adapt).

Modules

The terms "module" and "component" are often used interchangeably, which causes confusion. As I mentioned earlier, any system is composed of components. Therefore, a module is a component. However, not every component is a module. To design a flexible system, it's not enough to decompose the system into an arbitrary set of components. Instead, a modular design should enable you to alter the system by combining, reconfiguring, or replacing its components—its modules.

Let's consider two examples of modules from our everyday lives (Figure 4.1):

1. LEGO bricks are a straightforward illustration of modularity in action. Each brick is a self-contained unit that can be connected with other bricks to form a variety of structures. The ease with which these bricks can be assembled and disassembled illustrates a perfectly modular system.
2. Another widespread example of modularity is the interchangeable camera lenses used by photography enthusiasts. The ability to switch lenses enables photographers to adapt their cameras to different shooting conditions and achieve various effects, all without requiring multiple cameras.

The success of a modular system depends on the design of its modules. To enable the desired flexibility of a system, its design has to focus on clear boundaries and well-defined interactions between modules. To reason about the design of a module,

Figure 4.1 *Modularity in real-life systems*

(Images: left, focal point/Shutterstock; right, Kjpargeter/Shutterstock)

it's helpful to examine three fundamental properties describing a module: function, logic, and context:[2]

1. **Function** is the module's goal, the functionality it provides. It is exposed to consumers of the module through its public interface. The interface has to reflect the tasks that can be achieved by using the module, how the module can be integrated, and its interactions with other modules.
2. A module's **logic** is all about how the module's function is implemented; that is, the implementation details of the module. Unlike function, which is explicitly exposed to consumers, a module's logic should be hidden from other modules.
3. Finally, a module's **context** is the environment in which the module should be used. This includes both explicit requirements and implicit assumptions the design makes on the module's usage scenarios and environment.

These fundamental properties provide valuable insights into a module's role within the broader system, as summarized in Table 4.1.

To effectively design a module, its function should be clear and explicitly expressed in its public interface. The module's implementation details, or logic, on the other hand, should be hidden from consumers by the module's boundary. Ultimately, a clear and explicit definition of the context is essential for consumers to be able to integrate the module, as well as to be aware of how the module's behavior might be affected by changes in its environment.

Let's have a look at how these properties are reflected in the aforementioned modular systems: LEGO bricks and interchangeable camera lenses.

Table 4.1 *Comparison of the Three Fundamental Properties of a Module*

Property	Reflects	Type of information
Function	Module's goal	Public, explicit
Logic	How module works	Hidden by the module
Context	Assumptions about the environment	Public, less explicit than function

2. In later sources, you may encounter different terms used to describe the properties: border, implementation, and environment. For consistency, I'll stick to the original terminology (Myers 1979).

LEGO Bricks

The goal of the overall system—the LEGO constructor—is to form structures from individual building blocks. The modules of the system are LEGO bricks. As a module, each brick has the following properties:

- Function: A brick's goal is to connect with other bricks. It's explicitly reflected by the "integration interface": studs and holes through which it can be easily attached to other bricks.
- Logic: The bricks are made from a material that supports the required weight to build sturdy structures and guarantees reliable attachment to other bricks.
- Context: Since LEGOs are (generally) a toy for children, they have to be safe and appropriate for kids to play with. Furthermore, because of their purpose as a creative and fun playtime tool, using LEGO bricks to build actual houses would not be a good fit, as they are not designed or intended for such a task.

Camera Lenses

As I mentioned earlier in this chapter, interchangeable camera lenses enable photography enthusiasts to adapt to different shooting conditions without having to use multiple cameras. Both the camera body and the attachable lenses are modules. Let's focus on the properties of camera lenses as modules:

- Function: Enable capturing images with specific properties, such as focal length or aperture. The interface defines what kinds of cameras the lenses can be used with, as well as the supported range of optical capabilities.
- Logic: The inner workings of lenses allowing them to be connected to a camera and capture the required optical capabilities.
- Context: The supported ranges of camera models, as well as varying functionalities for different cameras (e.g., whether autofocus is supported or not).

Now that we have established an understanding of modularity in general, let's explore how these concepts apply in the context of software design.

Modularity in Software Systems

Although the term "module" is used extensively in software engineering, defining what a software module is, is not as straightforward as one might expect. The ambiguity arises from the term's long-standing use, during which its original meaning was obscured as software engineering evolved, leading to diverse reinterpretations and loss of a precise definition.

What makes a software module? Is it a library, a package, an object, a group of objects, or a service? Furthermore, what is a nonmodule software component, and how does it differ from a module?

Some argue that a module embodies a logical boundary, such as a namespace, a package, or an object, while a component signifies a physical boundary, encompassing artifacts such as services and redistributable libraries. However, the juxtaposition of logical and physical boundaries is not accurate. To understand why it's not accurate, as well as what exactly a software module is, let's go back in time and examine what was meant by "module" when the term was originally introduced to software design.

Software Modules

In his seminal paper "On the Criteria to Be Used in Decomposing Systems into Modules," David L. Parnas (1971) succinctly defined a module as "a responsibility assignment" rather than just an arbitrary boundary around statements of a program.

Four years later, in their book *Structured Design*, Edward Yourdon and Larry L. Constantine (1975) described a module as "a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier." Or, in simpler terms, a module is any collection of executable program statements meeting all of the following criteria (Myers 1979):

- The statements implement self-contained functionality.
- The functionality can be called from any other module.
- The implementation has the potential to be independently compiled.

The self-contained functionality criterion implies that a specific functionality is encapsulated within a module, rather than, for example, being spread across multiple modules. Next, the module makes this functionality accessible to other modules of the system through its public interface. Ultimately, the module's implementation

can *potentially* be independently compiled. Consequently, according to this definition, the type of a module's boundary—physical or logical—is not essential. As long as it has the potential of being extracted into an independent unit that can be compiled, it is a module. What is more important than the type of the module's boundary is the functionality it implements and provides to other modules.

This focus on the well-defined functionality rather than the type of a boundary makes modules ubiquitous all across software design. (Micro)services, frameworks, libraries, namespaces, packages, objects, classes—all can be modules. Furthermore, because nowadays a class's methods can be compiled independently,[3] even individual methods/functions can be considered modules.

That means a service-based system can be modular if its services are designed as effective modules. A service of that system can be modular on its own if, for example, it consists of modular namespaces. Modular objects can form a modular namespace, and the same is true for methods or functions constituting objects. "It's turtles all the way down," as illustrated in Figure 4.2. Modules are not flat; modular design is hierarchical.

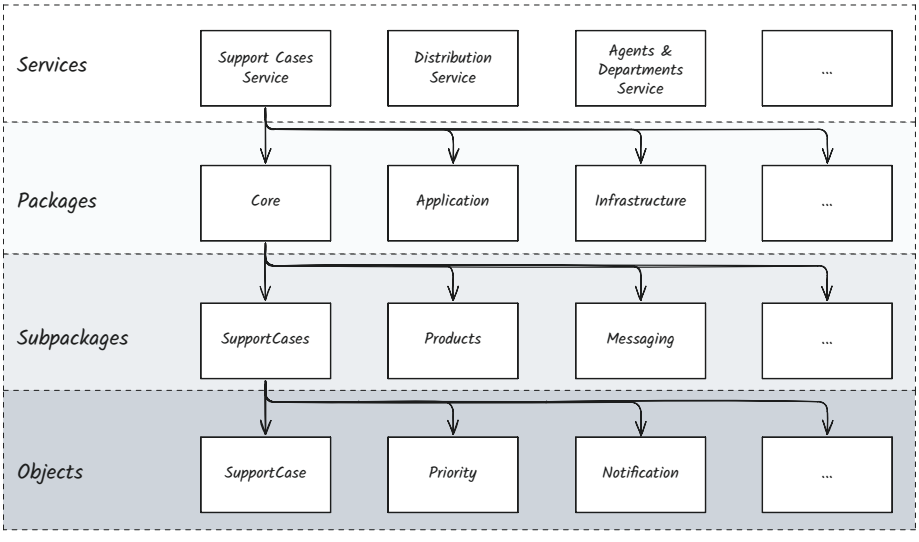


Figure 4.2 *Hierarchical modular design*

3. For example, extension methods in C# or functions in languages such as Python and JavaScript.

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).

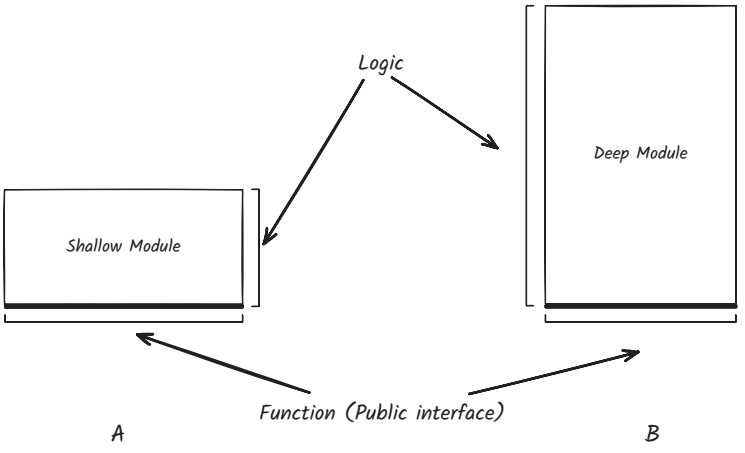


Figure 4.3 *A shallow module (A) and a deep module (B)*

According to Ousterhout, the resultant "depth" of the rectangle reflects how effective it is at hiding knowledge. The higher the ratio between the module's function and logic, the "deeper" the rectangle.

If the module is shallow, as in Figure 4.3A, the difference between the function and logic is small. That is, the complexity encapsulated by the module's boundary is low as well. In the extreme case, the function and logic are precisely the same—the public interface reflects how the module is implemented. Such an interface provides no value; it doesn't encapsulate any complexity. You could just as well read the module's implementation. Listing 4.2 shows an extreme example of a shallow module. The method's interface doesn't encapsulate any knowledge. Instead, it simply describes its implementation (adding two numbers).

Listing 4.2 *An Example of a Shallow Module*

```
addTwoNumbers(a, b) {
    return a + b;
}
```

On the other hand, a deep module (Figure 4.3B) encapsulates the complexity of the implementation details behind a concise public interface. The consumer of such a module need not be aware of its implementation details. Instead, the consumer can reason about the module's functionality and its role in the overarching system, while being ignorant of how the module is implemented—at a higher semantic level.

That said, the metaphor of deep modules has its limitations. For instance, there can be two perfectly deep modules implementing the same business rule. If this business rule changes, both modules will need to be modified. This could lead to cascading changes throughout the system, creating an opportunity for inconsistent system behavior if only one of the modules is updated. This underscores the hard truth about modularity: Confronting complexity is difficult.

Modularity Versus Complexity

Modularity and complexity are two competing forces. Modularity aims to make systems easier to understand and to evolve, while complexity pulls the design in the opposite direction.

The complete opposite of modularity, and the epitome of complexity, is the Big Ball of Mud anti-pattern (Foote and Yoder 1997):

> A Big Ball of Mud is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. —Brian Foote and Joseph Yoder

In the preceding definition of the Big Ball of Mud anti-pattern, unregulated growth, sharing information promiscuously among distant elements of the system, and important information becoming global or duplicated all demonstrate how unoptimized and inefficient flow of knowledge cripples systems.

These points can also be formulated in terms of ineffective abstractions. An effective abstraction removes all extraneous information, retaining only what is absolutely necessary for effective communication. In contrast, an ineffective abstraction creates noise by failing to eliminate unimportant details, removing essential details, or both.

If an abstraction includes extraneous details, it exposes more knowledge than is actually necessary. That causes accidental complexity in multiple ways. The consumers of the abstraction are exposed to more details than are actually needed to use the abstraction. First, this results in accidental cognitive load, or cognitive load that could have been avoided by encapsulating the extraneous detail. Second, this limits the scope of the abstraction: It is no longer able to represent a group of entities equally well, but only those for which the extraneous details are relevant.

On the other hand, an abstraction can fail if it omits important information. For example, a database abstraction layer that doesn't communicate its transaction semantics may result in users expecting a different level of data consistency than the

one provided by concrete implementation. This situation creates what is referred to as a leaking abstraction;[7] that is, when details from the underlying system "leak" through the abstraction. This happens when the consumer of the abstraction needs to understand the underlying concrete implementation to use it correctly. As in the case of an abstraction sharing extraneous details, it increases the consumer's cognitive load and can lead to misuse or misunderstandings of the module, complicating maintenance, debugging, and extension.

Hence, encapsulating knowledge is a double-edged sword. Going overboard can make it hard or even impossible to use the module, but the same is true when too little knowledge is being communicated. To make modularity even more challenging, even if a system is decomposed into seemingly perfect modules, it is still not guaranteed to be modular.

Modularity: Too Much of a Good Thing

In the beginning of the chapter, I defined modular design as one that allows the system to adapt to future changes. But how flexible should it be? As they say, the more reusable something is, the less useful it becomes. That is the cost of flexibility and, consequently, of modularity.

When designing a modular system, it's crucial to watch for the two extremes: not making a system so rigid that it can't change, and not designing a system to be so flexible that it's not usable. As an example, let's revisit the repository object in Listing 4.1. Its interface allows two ways of querying for customers: by name and by phone number. What would that interface look like if we were to attempt to address all possible query types; for example, by a wider range of fields, or even by looking up customers based on aggregations of values? That would make the interface much harder to work with. Moreover, optimizing the underlying database to efficiently handle all possible queries wouldn't be trivial.

Hence, a modular design should focus on *reasonable* changes. In other words, the system should expose only reasonable degrees of freedom. For example, changing the functionality of a blog engine into a printer driver is not a reasonable change.

Unfortunately, identifying reasonable future changes is not an exact science, but is based on our current knowledge and assumptions about the system. An assumption is essentially a bet against the future (Høst 2023). The future can support or

7. Spolsky, Joel. "The Law of Leaky Abstractions." Joel on Software. November 11, 2002. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions.

invalidate assumptions. However, if modular design entails making bets, we can gather as much information as possible and do our best to make informed bets.

Coupling in Modularity

Many aspects of modularity can be understood only by considering the modules not as individual entities, but by examining them in relation to one another. This was demonstrated in the Deep Modules section earlier: Even perfectly "deep" modules can still introduce complex interactions. As Alan Kay said, the big idea of object-oriented programming is messaging, not classes;[8] in other words, the relationships and interactions between objects.[9] Traditionally, when systems are designed the main focus is on the components, or boxes. But what about the arrows and lines connecting them?

The modularity of a system cannot be evaluated by examining designs of individual modules in isolation. The goal of modular design is to simplify the relationships between components of a system. Hence, modularity can only be evaluated in the scope of the relationships and interactions between the components. The knowledge that is shared among the components controls whether the overarching system will be more modular or more complex. Coupling is the aspect of a system that defines what knowledge is shared between components of a system. Different ways of coupling components share different types and amounts of knowledge. Some will increase complexity, while others will contribute to modularity.

This is a good time to mention the counterpart of coupling: cohesion. The concept of cohesion was introduced in tandem with coupling in *Structured Design* (Yourdon and Constantine 1975). Cohesion refers to the degree to which the elements inside a module belong together. In other words, it's a measure of how closely the responsibilities of a module are related to each other. High cohesion is generally seen as a desirable characteristic because it promotes a single, well-defined purpose for each module, improving understandability, maintainability, and robustness.

Under the hood, however, cohesion is based on coupling. Some software engineers even refer to cohesion as "good coupling." That's my preferred approach as well. The chapters in Part II, Dimensions, scrutinize how coupling affects system design and the dimensions in which effects of coupling can be observed. Later in the book I will combine these insights into a concise framework for guiding modular design that will also reflect cohesion of the system's modules.

8. Kay, Alan. "Alan Kay on Messaging." October 10, 1998. http://wiki.c2.com/?AlanKayOnMessaging.
9. According to the original definition, objects are modules and, therefore, the same design principles that apply to objects apply to modules at other levels of abstraction.

Key Takeaways

Modularity strives to minimize complexity by managing the distribution of knowledge across modules. However, the overarching goal of modularity is to enable evolution of the system according to future goals and needs. Hence, modular design requires awareness not only of the current requirements, but also of those that might arise in the future.

Nevertheless, be aware of the "too much of a good thing" syndrome. A system that is flexible to accommodate any change is likely to be overly complicated. Striking a balance is crucial to prevent systems from becoming exceedingly rigid or overly flexible.

To train your "muscle" of predicting future changes, learn about the business domain of your system. Analyze the trends: what changes were required in the past, and why. Learn about competing products: what they are doing differently, why they are doing these things differently, and how likely the functionality is to change in your system.

When designing modules, reason about their core properties. Can you state a module's function (purpose) without revealing its implementation details (logic)? Is a module's usage scenario (context) explicitly stated, or is it based on assumptions that might be forgotten over time?

Ultimately, to truly design modular systems, one must consider modules in relation to each other, acknowledging that their interplay significantly impacts modularity. Coupling defines what knowledge is shared between components, and cohesion indicates how related a module's responsibilities are. These topics will be expanded upon in the forthcoming chapters, eventually forming a robust framework to inform modular design.

Quiz

1. What are the basic properties of a module?
   a. API, database, and business logic
   b. Source code
   c. Function and logic
   d. Function, logic, and context

2. What makes an effective module?
   a. Runtime performance
   b. Maximizing the complexity it encapsulates
   c. Maximizing the complexity it encapsulates while supporting the system's flexibility needs
   d. Correct implementation of the business logic

3. Which property of a module is the most explicit?
   a. Function
   b. Logic
   c. Context
   d. Answers B and C are correct.

4. Which of the following software design elements can be considered modules?
   a. Services
   b. Namespaces
   c. Classes
   d. All of the answers are correct.

5. What makes an effective abstraction?
   a. Omitting as much information as possible
   b. Retaining as much detail as possible
   c. Creating a language that allows discussing about functionalities of components, without having to know how they are implemented
   d. Describing as many objects as possible

This page intentionally left blank

Index

D

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V

W

X-Y-Z