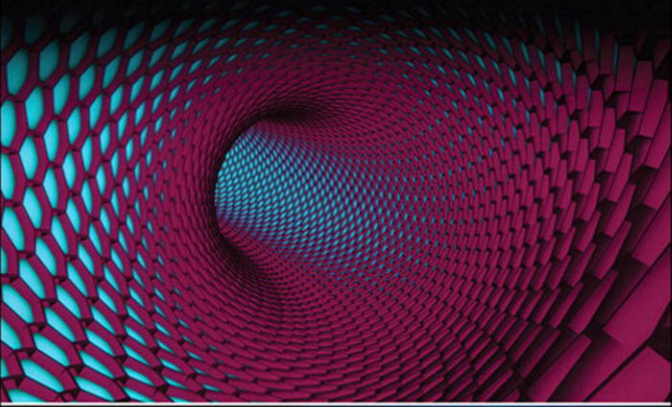
PRENTICE HALL

Robert C. Martin Series

Java Application Architecture

Modularity Patterns with Examples Using OSGi



Kirk Knoernschild

Forewords by Robert C. Martin *and* Peter Kriens

Praise for *Java Application Architecture*

"The fundamentals never go out of style, and in this book Kirk returns us to the fundamentals of architecting economically interesting software-intensive systems of quality. You'll find this work to be well-written, timely, and full of pragmatic ideas."

—Grady Booch, IBM Fellow

"Along with GOF's *Design Patterns*, Kirk Knoernschild's *Java Application Architecture* is a must-own for every enterprise developer and architect and on the required reading list for all Paremus engineers."

—Richard Nicholson, Paremus CEO, President of the OSGi Alliance

"In writing this book, Kirk has done the software community a great service: He's captured much of the received wisdom about modularity in a form that can be understood by newcomers, taught in computer science courses, and referred to by experienced programmers. I hope this book finds the wide audience it deserves."

—Glyn Normington, Eclipse Virgo Project Lead

"Our industry needs to start thinking in terms of modules—it needs this book!"

—Chris Chedgey, Founder and CEO, Structure 101

"In this book, Kirk Knoernschild provides us with the design patterns we need to make modular software development work in the real world. While it's true that modularity can help us manage complexity and create more maintainable software, there's no free lunch. If you want to achieve the benefits modularity has to offer, buy this book."

—Patrick Paulin, Consultant and Trainer, Modular Mind

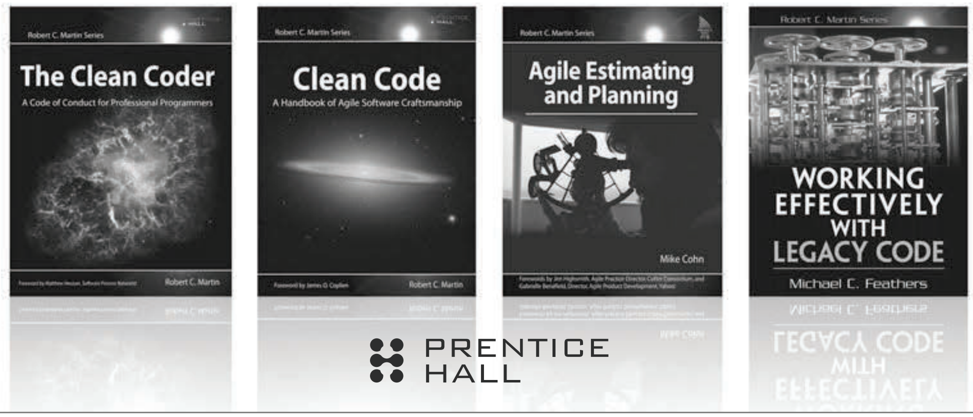
"Kirk has expertly documented the best practices for using OSGi and Eclipse runtime technology. A book any senior Java developer needs to read to better understand how to create great software."

—Mike Milinkovich, Executive Director, Eclipse Foundation

This page intentionally left blank

Java Application Architecture

The Robert C. Martin Series


Robert C. Martin Series
The Clean Coder
A Code of Conduct for Professional Programmers
Robert C. Martin
Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin
Agile Estimating and Planning
Mike Cohn
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathers
PRENTICE HALL

Visit **informit.com/martinseries** for a complete list of available publications.

The Robert C. Martin Series is directed at software developers, team-leaders, business analysts, and managers who want to increase their skills and proficiency to the level of a Master Craftsman. The series contains books that guide software professionals in the principles, patterns, and practices of programming, software project management, requirements gathering, design, analysis, testing and others.

PRENTICE HALL informIT.com | Safari Books Online

ALWAYS LEARNING PEARSON

Java Application Architecture

Modularity Patterns with Examples Using OSGi

Kirk Knoernschild

PRENTICE
HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data
Knoernschild, Kirk.
Java application architecture : modularity patterns with examples using OSGi / Kirk Knoernschild.
p. cm.
Includes index.
ISBN 978-0-321-24713-1 (pbk. : alk. paper)
1. Java (Computer program language) 2. Application software—Development. 3. Software architecture. 4. Component software. I. Title.
QA76.73.J38K563 2012
005.13'3—dc23

2011051434

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-24713-1
ISBN-10: 0-321-24713-2
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, March 2012

Tammy,
My wife, best friend, and soul mate . . . forever!
Thank you for all that you do.

I love you.

Cory,
Fly high.

Cody,
Play ball.

Izi,
Cheer loud.

Chloe,
Dream big.

This page intentionally left blank

Contents

Foreword by Robert C. Martin

I'm dancing! By God I'm dancing on the walls. I'm dancing on the ceiling. I'm ecstatic. I'm overjoyed. I'm really, really pleased.

"Why?" you ask. Well, I'll tell you why—since you asked. I'm happy because *somebody finally read John Lakos's book!*

Way back in the 1990s, John Lakos wrote a book entitled *Large-Scale C++ Software Design*. The book was brilliant. The book was groundbreaking. The book made the case for large-scale application architecture and made it well.

There was just one problem with John's book. The book had "C++" in the title and was published just as the software community was leaping to Java. And so the people who really needed to read that book didn't read it.

Ah, but then the people doing Java back then weren't reading *any* books on software design, because they were all 22 years old, sitting in Herman-Miller office chairs, hacking Java, day trading, and dreaming of being billionaires by the time they were 23. Oh, God, they were such hot stuff!

So, here we are, more than a decade later. We've matured a bit. And we've failed a bit. And our failures have winnowed and seasoned us. We now look back at the wasteland of Java architectures we created and grimace. How could we have been so naïve? How could we have lost sight of the principles of Jacobson, Booch, Rumbaugh, Fowler, and Lakos? Where did we go wrong?

I'll tell you where we went wrong. The Web bamboozled us. We all got Twitterpated. We thought the Web was revolutionary. We thought the

Web changed everything. We thought the Web made all the old rules irrelevant. We thought the Web was so new, so revolutionary, and so game-changing that we ignored the rules of the game.

And we paid. Oh, God, how we paid. We paid with huge, unmanageable designs. We paid with tangled, messy code. We paid with misguided directionless architectures. We paid with failed projects, bankrupt companies, and broken dreams. We paid, and we paid, and we paid.

It took 15 years, and we've just begun to realize why. We've just begun to see that the game hasn't changed at all. We've begun to see that the Web is just another delivery mechanism, no different from all the others—a reincarnation of the old IBM green-screen request/response technology. It was just plain old software after all, and we should never have abandoned the rules of the game.

Now we can see that we should have stuck to the wisdom of Parnas, Weinberg, Page-Jones, and DeMarco all along. We should never have walked away from the teachings of Jacobson and Booch. *And we should have read that damn book by Lakos!*

Well, somebody *did* read that book. And he must have read a few others, too, because he's written a book that states the rules of the Java architecture game better than I've seen them stated before. You're holding that book in your hands right now. The man who wrote it is named Kirk Knoernschild.

In this book Kirk has gone beyond Lakos, beyond Jacobson, beyond Booch. He's taken the principles of those past masters and created a brilliant new synthesis of principles, rules, guidelines, and patterns. *This is how you build a Java application, people.*

Go ahead and flip through the pages. Notice something? Yeah, no fluff! It's all hard-core. It's all right to the point. It's all pragmatic, useful, necessary! It's all about the nuts and bolts architecture of Java applications—the way it *should* be: modular, decoupled, levelized, independently deployable, and smart.

If you are a Java programmer, if you are a tech lead or a team lead, if you are an architect, if you are someone who wants and needs to make a difference on your software development team, *read this book.* If you want to avoid repeating the tragedy of the last 15 years, *read this book.* If you want to learn what software architecture is really all about, *read this book!*

Nuff said.

—Uncle Bob
35,000 feet over the Atlantic
October 1, 2011

Foreword by Peter Kriens

About two years ago (January 2010) I got an e-mail from Kirk Knoernschild, soliciting feedback for his almost-ready book. Looking back at the heated discussion that ensued—50 or more lengthy mails—I cannot but wonder that some resentment must have formed on his side. I am pretty sure our conversations caused heavy delays in his initial schedule. I was therefore pleasantly surprised when Kirk asked me to write a foreword for this book; it takes a strong man to let an opponent write a foreword for the book he put so much effort into.

Now, I do agree with most of what Kirk says in this book. We are both intrigued by the magic of modularity, and we see eye to eye on most of the fundamental concepts. However, as is so often the case, the most heated debates are between people who agree on the principles but differ on the details. It was not until the OSGi Community Event in Darmstadt, Germany (two days before the deadline of this foreword), that I suddenly understood Kirk's resistance.

At this event Graham Charters (IBM) presented the "Modularity Maturity Model," which he derived from IBM's SOA Maturity Model, which of course came from the original SEI Capability Maturity Model (CMM). This was an insightful presentation that made me understand that my perspective of system design is very much tainted by more than 13 years of living modularity.

One of the key lessons of the CMM was that it is impossible to skip a step. If your company is on level 1 of CMM (chaotic), then it is not a good idea to make plans to move to level 4 (managed) in one giant step. Companies have tried and failed spectacularly. Transitions through each of the intermediate stages are required to help organizations understand the intricacies of the different levels. Every level has its own set of problems that are solved by the next level.

After Graham's presentation, it became clear to me that I basically look down from level 5, and Kirk is trying to make people look up from level 1. The particular issues that we were disagreeing on are about the challenges you will encounter when designing modular software. These challenges seem perfectly sensible after you've reached level 2 or 3 but tend not to make a lot of sense on level 1. Our brains are wired in such a way that we can understand a solution only once we experience the corresponding problem. I was trying to beat Kirk into discussing those solutions before his readers had experienced and understood the problems of the prior levels.

In my Modularity Maturity Model (Graham's was a bit different), I see the following levels:

1. Unmanaged/chaos
2. Managing dependencies
3. Proper isolation
4. Modifying the code base to minimize coupling
5. Service-oriented architectures

In the first level, applications are based on the class path, a linear list of JARs. Applications consist of a set of JARs or directories with classes that form the classpath. In this level there is no modularity whatsoever. Problems on this level are missing classes or mixing versions.

The second level is when you get module identity and specify dependencies on other modules. Modules get a name and can be versioned. They still are linearly searched, and many of the problems from level 1 exist, but the system is more maintainable and the results more repeatable. Problems on this level mainly circulate around "downloading the Internet" because of excessive transitive dependencies. This is the level Maven is currently at.

The third stage is to truly isolate the modules from each other with a very distinct set of exported, imported, and private packages. Dependencies can now be expressed on packages, reducing the need to "download the Internet." This isolation provides an internal namespace for a module that is truly local to the module, and it allows multiple namespaces so that different versions of a package can be supported in the same system. The problems at this level are usually caused by popular Java patterns based on dynamic class loading that are rarely compatible with module boundaries and multiple namespaces.

The fourth level starts when the code base is modified only for the purpose of maximizing cohesion and minimizing coupling. There is increased awareness that a single line in the code can actually cause an excessive amount of dependencies. Combining or separating functions can have a significant influence on how the system behaves during deployment. At this level the existing Java patterns become painful to use because they often require central configuration, while the solutions seem to indicate a more equal peer-to-peer model. In OSGi, µServices become very attractive since they solve many problems.

At level 5, the last level, the modules become less important than the µServices they provide. Design and dependency resolution is now completely by µServices; modules are just containers that consume and provide µServices.

In the past 13 years I've lived and breathed level 5 as it is implemented in OSGi. This sometimes makes it hard to empathize with people who have used only the classpath and simple JARs. Looking at Graham Charter's presentation, I realized that Kirk's ambition is to help people understand the importance of modular design principles and move them from level 1 to level 2, ultimately giving them a solid foundation to achieve even greater maturity with OSGi. I realize that I often tried to drag the book straight to level 5, foregoing several important lessons that are necessary to design modular software. That book is still critical and is one I hope to write myself someday.

Kirk's book is so important now because it provides patterns to get started with modular thinking and allows you to begin your journey in building modular software using the platforms, frameworks, and languages most widely used today. Yes, I do believe there are better solutions to some of the problems in this book, but I also realize that better is often the enemy of good.

This is therefore an excellent book if you build Java applications using Spring, Guice, or other popular dependency injection frameworks but continue to experience the pain of brittle and rigid software that is difficult and expensive to maintain. The global coupling of your code makes it hard to add new functionality or change the existing code base. This book teaches you many of the fundamental lessons of modularity and will give you a view into the magic of modularity.

That said, I also hope you pay special attention to the examples throughout the book that use OSGi. The first is at the end of Chapter 4 and demonstrates how OSGi helps you achieve proper isolation and minimize coupling using µServices. As much value as this book provides, I am convinced that following its advice will help you build software with greater architectural integrity and will lead you on the correct migration path toward OSGi. OSGi is by far the most mature modularity solution around.

Kirk has been a more than worthy opponent; he has taught me more about my own ideas than almost anybody else in the last few years by forcing me to put them into words. I do hope you will have as much fun reading this book as I had discussing this book with him over the last two years.

—Peter Kriens
Technical Director, OSGi Alliance
Beaulieu, England
September 2011

Acknowledgments

The inspiration for this book comes from several sources, and the help I've received over the past several years is tremendous. However, I owe a very special thanks to seven individuals. It is their ideas that have guided my work over the past two decades, the development of these patterns over the past ten years, and the completion of this book over the past two years. They include the following:

Robert C. Martin (*Uncle Bob*): Bob's work on object-oriented design principles (i.e., the SOLID principles) is a cornerstone of many of the techniques discussed throughout this book. In fact, this book is part of his series, and Appendix A provides an overview of several of the principles.

Clemens Szyperski: Clemens's *Component Software: Beyond Object Oriented Programming* served as the building block upon which the definition of *module* is used throughout this book.

John Lakos: Johns's *Large Scale C++ Software Design* is the only book I'm aware of that discusses physical design. The ideas in John's book served as inspiration and increased my interest in physical design, allowing me, over the past ten years, to apply and refine techniques that have resulted in the modularity patterns.

Ralph Johnson, John Vlissides, Erich Gamma, and Richard Helm ("the GOF" or "the Gang of Four"): Aside from providing the

pattern template I use throughout this book, *Design Patterns* helped cement my understanding of object-oriented concepts.

Additionally, I want to thank the following individuals whose feedback has served me tremendously in helping improve the book's message.

Notably, Peter Kriens, technology director of the OSGi Alliance: Peter provided enough feedback that I should have probably listed him as a coauthor.

I'd also like to thank Brad Appleton, Kevin Bodie, Alex Buckley, Robert Bogetti, Chris Chedgey, Michael Haupt, Richard Nicholson, Glyn Normington, Patrick Paulin, John Pantone, and Vineet Sinha for providing thoughtful reviews and valuable feedback that helped me clarify certain areas of the text and provide alternative views on the discussion. Of course, along this journey, several others have influenced my work. Sadly, I'm sure I've neglected to mention a few of them. You know who you are. Thank you!

Of course, the Prentice Hall team helped make it all happen. Chris Guzikowski, my editor, gave me more chances over the past several years to complete this book than I probably deserved. Sheri Cain, my development editor, provided valuable formatting advice, answered several of my silly questions, and helped me structure and refine a very rough manuscript. Olivia Basegio and Raina Chrobak, the editorial assistants, helped guide me through the entire process. Anna Popick, the project editor, saw it through to completion. And Kim Wimpsett, my copy editor, helped polish the final manuscript.

Finally, I want to thank my family. Without their love, few things are possible, and nothing is worthwhile. Mom and Dad, for their gentle guidance along life's journey. I'm sure there were many times they wondered where I was headed. Grandma Maude, the greatest teacher there ever was. My children, Cory, Cody, Izi, and Chloe, who make sure there is never a dull moment. And of course, my wife, Tammy. My best friend whose encouragement inspired me to dust off an old copy of the manuscript and start writing again. Thank you. All of you!

About the Author

Kirk Knoernschild is a software developer who has filled most roles on a software development team. Kirk is the author of *Java Design: Objects, UML, and Process* (Addison-Wesley, 2002), and he contributed to *No Fluff Just Stuff 2006 Anthology* (Pragmatic Bookshelf, 2006). Kirk is an open source contributor, has written numerous articles, and is a frequent conference speaker. He has trained and mentored thousands of software professionals on topics including Java/J2EE, modeling, software architecture and design, component-based development, service-oriented architecture, and software process. You can visit his website at http://techdistrict.kirkk.com.

This page intentionally left blank

Introduction

In 1995, design patterns were all the rage. Today, I find the exact opposite. Patterns have become commonplace, and most developers use patterns on a daily basis without giving it much thought. New patterns rarely emerge today that have the same impact of the Gang of Four (GOF) patterns.[1] In fact, the industry has largely moved past the patterns movement. Patterns are no longer fashionable. They are simply part of a developer's arsenal of tools that help them design software systems.

But, the role design patterns have played over the past decade should not be diminished. They were a catalyst that propelled object-oriented development into the mainstream. They helped legions of developers understand the real value of inheritance and how to use it effectively. Patterns provided insight into how to construct flexible and resilient software systems. With nuggets of wisdom, such as "Favor object composition over class inheritance" and "Program to an interface, not an implementation" (Gamma 1995), patterns helped a generation of software developers adopt a new programming paradigm.

Patterns are still widely used today, but for many developers, they are instinctive. No longer do developers debate the merits of using the Strategy pattern. Nor must they constantly reference the GOF book to identify

1. The patterns in the book *Design Patterns: Elements of Reusable Object-Oriented Software* are affectionately referred to as the GOF patterns. GOF stands for the Gang of Four, in reference to the four authors.

which pattern might best fit their current need. Instead, good developers now instinctively design object-oriented software systems.

Many patterns are also timeless. That is, they are not tied to a specific platform, programming language, nor era of programming. With some slight modification and attention to detail, a pattern is molded to a form appropriate given the context. Many things dictate context, including platform, language, and the intricacies of the problem you're trying to solve. As we learn more about patterns, we offer samples that show how to use patterns in a specific language. We call these *idioms*.

I'd like to think the modularity patterns in this book are also timeless. They are not tied to a specific platform or language. Whether you're using Java or .NET, OSGi,[2] or Jigsaw[3] or you want to build more modular software, the patterns in this book help you do that. I'd also like to think that over time, we'll see idioms emerge that illustrate how to apply these patterns on platforms that support modularity and that tools will emerge that help us refactor our software systems using these patterns. I'm hopeful that when these tools emerge, they will continue to evolve and aid the development of modular software. But most important, I hope that with your help, these patterns will evolve and morph into a pattern language that will help us design better software—software that realizes the advantages of modularity. Time will tell.

Object-Oriented Design

SOLID principles, 319

Over the past several years, a number of object-oriented design principles have emerged. Many of these design principles are embodied within design patterns. The SOLID design principles espoused by Uncle Bob are prime examples. Further analysis of the GOF patterns reveals that many of them adhere to these principles.

For all the knowledge shared, and advancements made, that help guide object-oriented development, creating very large software systems is still inherently difficult. These large systems are still difficult to maintain, extend, and manage. The current principles and patterns of object-oriented development fail in helping manage the complexity of large software

2. OSGi is the dynamic module system for the Java platform. It is a specification managed by the OSGi Alliance. For more, see www.osgi.org.

3. Jigsaw is the proposed module system for Java SE 8.

systems because they address a different problem. They help address problems related to logical design but do not help address the challenges of physical design.

Logical versus Physical Design

Almost all principles and patterns that aid in software design and architecture address logical design.[4] Logical design pertains to language constructs such as classes, operators, methods, and packages. Identifying the methods of a class, relationships between classes, and a system package structure are all logical design issues.

It's no surprise that because most principles and patterns emphasize logical design, the majority of developers spend their time dealing with logical design issues. When designing classes and their methods, you are defining the system's logical design. Deciding whether a class should be a Singleton is a logical design issue. So is determining whether an operation should be abstract or deciding whether you should inherit from a class versus contain it. Developers live in the code and are constantly dealing with logical design issues.

Making good use of object-oriented design principles and patterns is important. Accommodating the complex behaviors required by most business applications is a challenging task, and failing to create a flexible class structure can have a negative impact on future growth and extensibility. But logical design is not the focus of this book. Numerous other books and articles provide the guiding wisdom necessary to create good logical designs. Logical design is just one piece of the software design and architecture challenge. The other piece of the challenge is physical design. If you don't consider the physical design of your system, then your logical design, no matter how beautiful, may not provide you with the benefits you believe it does. In other words, logical design without physical design may not really matter all that much.

Physical design represents the physical entities of your software system. Determining how a software system is packaged into its deployable units is a physical design issue. Determining which classes belong in

4. One exception is the excellent book by John Lakos, *Large-Scale C++ Software Design*. Here, Lakos presents several principles of logical and physical design to aid development of software programs written using C++.

which deployable units is also a physical design issue. Managing the relationships between the deployable entities is also a physical design issue. Physical design is equally as, if not more important than, logical design.

For example, defining an interface to decouple clients from all classes implementing the interface is a logical design issue. Decoupling in this fashion certainly allows you to create new implementations of the interface without impacting clients. However, the allocation of the interface and its implementing classes to their physical entities is a physical design issue. If the interface has several different implementations and each of those implementation classes has underlying dependencies, the placement of the interface and implementation has a tremendous impact on the overall quality of the system's software architecture. Placing the interface and implementation in the same module introduces the risk of undesirable deployment dependencies. If one of the implementations is dependent upon a complex underlying structure, then you'll be forced to include this dependent structure in all deployments, regardless of which implementation you choose to use. Regardless of the quality of the logical design, the dependencies between the physical entities will inhibit reusability, maintainability, and many other benefits you hope to achieve with your design.

Unfortunately, although many teams spend a good share of time on logical design, few teams devote effort to their physical design. Physical design is about how we partition the software system into a system of modules. Physical design is about software modularity.

MODULARITY

Large software systems are inherently more complex to develop and maintain than smaller systems. Modularity involves breaking a large system into separate physical entities that ultimately makes the system easier to understand. By understanding the behaviors contained within a module and the dependencies that exist between modules, it's easier to identify and assess the ramification of change.

For instance, software modules with few incoming dependencies are easier to change than software modules with many incoming dependencies. Likewise, software modules with few outgoing dependencies are much easier to reuse than software modules with many outgoing dependencies. Reuse and maintainability are important factors to consider when designing software modules, and dependencies play an important factor. But dependencies aren't the only factor.

Module cohesion also plays an important role in designing high-quality software modules. A module with too little behavior doesn't do enough to be useful to other modules using it and therefore provides minimal value. Contrarily, a module that does too much is difficult to reuse because it provides more behavior than other modules desire. When designing modules, identifying the right level of granularity is important. Modules that are too fine-grained provide minimal value and may also require other modules to be useful. Modules that are too coarse-grained are difficult to reuse.

The principles in this book provide guidance on designing modular software. They examine ways that you can minimize dependencies between modules while maximizing a module's reuse potential. Many of these principles would not be possible without the principles and patterns of object-oriented design. As you'll discover, the physical design decisions you make to modularize the system will often dictate the logical design decisions.

Unit of Modularity: The JAR File

Physical design on the Java platform is done by carefully designing the relationships and behavior of Java JAR files. On the Java platform, the unit of modularity is the JAR file. Although these principles can be applied to any other unit, such as packages, they shine when using them to design JAR files.

module defined, 17

OSGi

The OSGi Service Platform is the dynamic module system for Java. In OSGi parlance, a module is known as a *bundle*. OSGi provides a framework for managing bundles that are packaged as regular Java JAR files with an accompanying manifest. The manifest contains important metadata that describes the bundles and its dependencies to the OSGi framework.

OSGi, 273

You'll find examples leveraging OSGi throughout this book. However, OSGi is not a prerequisite for using the modularity patterns. OSGi simply provides a runtime environment that enables and enforces modularity on the Java platform. OSGi offers the following capabilities:

- **Modularity**: Enables and enforces a modular approach to architecture on the Java platform.
- **Versioning**: Supports multiple versions of the same software module deployed within the same Java Virtual Machine (JVM) instance.

- **Hot deployments**: Permits modules to be deployed and updated within a running system without restarting the application or the JVM.
- **Encapsulation**: Allows modules to hide their implementation details from consuming modules.
- **Service orientation**: Encourages service-oriented design principles in a more granular level within the JVM. To accomplish this, OSGi uses µServices.
- **Dependency management**: Requires explicit declaration of dependencies between modules.

Who This Book Is For

This book is for the software developer or architect responsible for developing software applications. If you're interested in improving the design of the systems you create, this book is for you.

This book is not exclusively for individuals who are using a platform that provides native support for modularity. For instance, if you're using OSGi, this book helps you leverage OSGi to design more modular software. But if you're not using OSGi, the techniques discussed in this book are still valuable in helping you apply techniques that increase the modularity of your software systems. Nor is this book exclusively for Java developers. Although the examples throughout this book use Java, the techniques discussed can be applied to other platforms, such as .NET, with relative ease.

If you want to understand more deeply the benefits of modularity and start designing modular software systems, this book is for you! This book provides answers to the following questions:

- What are the benefits of modularity and why is it important?
- How can I convince other developers of the importance of modularity?
- What techniques can I apply to increase the modularity of my software systems?
- How can I start using modularity now, even if I'm not developing on a platform with native support for modularity, such as OSGi?
- How can I migrate large-scale monolithic applications to applications with a modular architecture?

How This Book Is Organized

This book is divided into three parts. Part I presents the case for modularity. Here, you explore the important role that software modularity plays in designing software systems and learn why you want to design modular software. Part II is a catalog of 18 patterns that help you design more modular software. These patterns rely heavily on the ideas discussed in Part I. Part III introduces OSGi and demonstrates how a software system designed using the patterns in this book is well positioned to take advantage of platform support for modularity. Part III relies heavily on code examples to demonstrate the points made.

Naturally, I suggest reading the book cover to cover. But, you might also want to explore the book by jumping from chapter to chapter. Feel free! Throughout this book, in the margin, you'll notice several forward and backward references to the topics relevant to the current topic. This helps you navigate and consume the ideas more easily. The following is a summary of each chapter.

Part I: The Case for Modularity

Part I presents the reasons why modularity is important. It is the case for modularity. A brief synopsis of each chapter in Part I follows:

- **Chapter 1, "Module Defined":** This chapter introduces modularity and formally defines and identifies the characteristics of a software module. I encourage everyone to read this short chapter.
- **Chapter 2, "The Two Facets of Modularity":** There are two aspects to modularity: the runtime model and the development model. Much emphasis has been placed on providing runtime support for modularity. As more platforms provide runtime support for modularity, the importance of the development model will take center stage. The development model consists of the programming model and the design paradigm.
- **Chapter 3, "Architecture and Modularity":** Modularity plays a critical role in software architecture. It fills a gap that has existed since teams began developing enterprise software systems. This chapter examines the goal of software architecture and explores the important role modularity plays in realizing that goal.

- **Chapter 4, "Taming the Beast Named Complexity":** Enterprise software systems are fraught with complexity. Teams are challenged by technical debt, and systems are crumbling from rotting design. This chapter explains how modularity helps us tame the increasing complexity of software systems.
- **Chapter 5, "Realizing Reuse":** Reuse is the panacea of software development. Unfortunately, few organizations are able to realize high rates of reuse. This chapter examines the roadblocks that prevent organizations from realizing reuse and explores how modularity increases the chance of success.
- **Chapter 6, "Modularity and SOA":** Modularity and SOA are complementary in many ways. This chapter explores how modularity and SOA are a powerful combination.
- **Chapter 7, "Reference Implementation":** It's important to provide some decent samples that illustrate the concepts discussed. The reference implementation serves two purposes. First, it ties together the material in the first six chapters so you can see how these concepts are applied. Second, it lays the foundation for many of the patterns discussed in Part II.

PART II: THE PATTERNS

The patterns are a collection of modularity patterns. They are divided into five separate categories, each with a slightly different purpose. There is some tension between the different categories. For instance, the usability patterns aim to make it easy to use a module while the extensibility patterns make it easier to reuse modules. This tension between use and reuse is further discussed in Chapter 5.

- **Chapter 8, "Base Patterns":** The base patterns are the fundamental elements upon which many of the other patterns exist. They establish the conscientious thought process that go into designing systems with a modular architecture. They focus on modules as the unit of reuse, dependency management, and cohesion. All are important elements of well-designed modular software systems.
- **Chapter 9, "Dependency Patterns":** I've personally found it fascinating that development teams spend so much time designing class

relationships but spend so little time creating a supporting physical structure. Here, you find some guidance that helps you create a physical structure that emphasizes low coupling between modules. You'll also find some discussion exploring how module design impacts deployment.

- **Chapter 10, "Usability Patterns":** Although coupling is an important measurement, cohesion is equally important. It's easy to create and manage module dependencies if I throw all of my classes in a couple of JAR files. But in doing so, I've introduced a maintenance nightmare. In this chapter, we see patterns that help ensure our modules are cohesive units. It's interesting that you'll find some contention between the dependency patterns and usability patterns. I talk about this contention and what you can do to manage it.
- **Chapter 11, "Extensibility Patterns":** A goal in designing software systems is the ability to extend the system without making modifications to the existing codebase. Abstraction plays a central role in accomplishing this goal, but simply adding new functionality to an existing system is only part of the battle. We also want to be able to deploy those new additions without redeploying the entire application. The extensibility patterns focus on helping us achieve this goal.
- **Chapter 12, "Utility Patterns":** The utility patterns aid modular development. Unlike the other patterns, they don't emphasize reuse, extensibility, or usability. Instead, they discuss ways that modularity can be enforced and that help address quality-related issues.

Part III: POMA and OSGi

Standard Java gives you everything you need to begin using the patterns in this book. Undoubtedly, though, you want to see the patterns in the context of an environment that provides first-class support for modularity. In this section, we do just that and use the OSGi framework to illustrate this through example.

- **Chapter 13, "Introducing OSGi":** This chapter provides a brief introduction to OSGi, including its capabilities and benefits. This chapter isn't meant as a tutorial and assumes some cursory knowledge of OSGi. We talk about OSGi and modularity, including µServices and the Blueprint specification. Additionally, you'll see how the dynamism of OSGi brings modularity to the runtime environment. Finally, we wrap up by exploring how the patterns relate to development in OSGi. We point out how OSGi makes it easier to use some of the modularity patterns in their purest form.

- **Chapter 14, "The Loan Sample and OSGi":** As you read through the pattern discussions, you'll notice a common example we use is a loan system. In this chapter, we again use the loan system but refactor the application so that it runs in an OSGi environment. You'll be surprised that once you have a modular architecture, OSGi is just a simple step away.
- **Chapter 15, "OSGi and Scala":** The Java platform supports multiple languages, and OSGi doesn't inhibit you from using alternative languages on the Java platform. In this section, we show how we can create a Scala module and plug it into a system. You'll see two simple advantages. First, the modular architecture makes it easy to add code without making modifications to any other code in the system. Second, it clearly illustrates the dynamism of OSGi.
- **Chapter 16, "OSGi and Groovy":** Like the Scala example in Chapter 15, we develop another module using the Groovy programming language to further illustrate the flexibility and dynamicity of a runtime environment that supports modularity.
- **Chapter 17, "Future of OSGi":** What's the future of modularity and OSGi? How might it transform how we currently think about large enterprise software systems? In this chapter, we explore that future with a provocative look at what's in store for modularity and OSGi.

Pattern Form

Each pattern is consistent in structure to help maximize its readability. Each is also accompanied by an example that illustrates how the underlying principles it captures are applied. Not all sections appear for all patterns. In some cases, certain sections are omitted when a previous discussion can be referenced. The general structure of each pattern resembles the Gang of Four (GOF) format, which is the format used in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, structured as follows:

PATTERN NAME

First, the name of the pattern is presented. The name is important, because it helps establish a common vocabulary among developers.

PATTERN STATEMENT

The pattern statement is a summary that describes the pattern. This statement helps establish the intent of the pattern.

SKETCH

A sketch is a visual representation that shows the general structure of the pattern. Usually, the Unified Modeling Language (UML) is used here.

DESCRIPTION

The description offers a more detailed explanation of the problem that the pattern solves. The description establishes the motivation behind the pattern.

IMPLEMENTATION VARIATIONS

As with any pattern, subtle implementation details quickly arise when applying the pattern to a real-world problem. "Implementation Variations" discusses some of the more significant alternatives you should consider when applying the pattern.

CONSEQUENCES

All design decisions have advantages and disadvantages, and like most advice on software design, the use of these patterns must be judicious. While they offer a great deal of flexibility, that flexibility comes with a price. The "Consequences" section discusses some of the interesting things you'll likely encounter when applying the pattern and some of the probable outcomes should you decide to ignore the pattern. After reading through the consequences, you should have a better idea of when you'll want to apply the pattern and when you may want to consider using an alternative approach. Boiled down, this section represents the advantages and disadvantages of using the pattern, the price you'll pay, and the benefits you should realize.

SAMPLE

It's usually easier to understand a pattern when you can see a focused example. In this section, we walk through a sample that illustrates how the pattern can be applied. Sometimes, we work through some code, and other times, some simple visuals clearly convey the message. Most important though is that the sample won't exist in a vacuum. When we apply patterns in the real world, patterns are often used in conjunction with each other to create a more flexible tailored solution. In cases where it makes sense, the sample builds on previous samples illustrated in other patterns. The result is insight into how you can pragmatically apply the pattern in your work.

WRAPPING UP

This section offers a few closing thoughts on the pattern.

PATTERN CATALOG

The following are the modularity patterns:

- **Base Patterns**
  - Manage Relationships: Design module relationships.
  - Module Reuse: Emphasize reusability at the module level.
  - Cohesive Modules: Module behavior should serve a singular purpose.
- **Dependency Patterns**
  - Acyclic Relationships: Module relationships must be acyclic.
  - Levelize Modules: Module relationships should be levelized.
  - Physical Layers: Module relationships should not violate the conceptual layers.
  - Container Independence: Modules should be independent of the runtime container.

  - Independent Deployment: Modules should be independently deployable units.
- **Usability Patterns**
  - Published Interface: Make a module's published interface well known.
  - External Configuration: Modules should be externally configurable.
  - Default Implementation: Provide modules with a default implementation.
  - Module Facade: Create a facade serving as a coarse-grained entry point to another fine-grained module's underlying implementation.
- **Extensibility Patterns**
  - Abstract Module: Depend upon the abstract elements of a module.
  - Implementation Factory: Use factories to create a module's implementation classes.
  - Separate Abstractions: Place abstractions and the classes that implement them in separate modules.
- **Utility Patterns**
  - Colocate Exceptions: Exceptions should be close to the class or interface that throws them.
  - Levelize Build: Execute the build in accordance with module levelization.
  - Test Module: Each module should have a corresponding test module.

THE CODE

Numerous examples are spread throughout this book, and many of these samples include code. All pattern samples for this book can be found in the following GitHub repository: https://github.com/pragkirk/poma.

If you're interested in running the code on your machine but are unfamiliar with Git, see the Git documentation at http://git-scm.com/documentation.

The sample code in Chapter 7 can be found in a Google Code Subversion repository at http://code.google.com/p/kcode/source/browse/#svn/trunk/billpayevolution/billpay.

I encourage everyone to download the code from these repositories and use the code while reading each pattern's "Sample" section. Although code is included with many of the patterns, it's not possible to include all the code for each sample. The code you find in this book helps guide you through the discussion and provides an overview of how the pattern can be applied. But, you gain far greater insight to the intricacies of the pattern by downloading and reviewing the code.

An Opening Thought on the Modularity Patterns

There was some debate surrounding the modularity patterns as I wrote this book. Some suggested they would be more aptly referred to as principles, while others preferred laws. Some even suggested referring to them as heuristics, guidelines, idioms, recipes, or rules. At the end of the day, however, all reviewers said they loved this book's content and approach. So, in the end, I stuck with patterns. Instead of trying to decide whether you feel these should be patterns, principles, heuristics, or something else, I encourage you to focus on the topic of discussion for each pattern. The idea! That's what's important.

Reference

Gamma, Erich, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

3 Architecture and Modularity

Modularity plays an important role in software architecture. It fills a gap that has existed since we began developing enterprise software systems in Java. This chapter discusses that gap and explores how modularity is an important intermediary technology that fills that gap.

3.1 Defining Architecture

There are numerous definitions of *architecture*. But within each lies a common theme and some key phrases. Here are a few of the definitions. From Booch, Rumbaugh, and Jacobson (1999):

> *An architecture is the set of* ***significant decisions about the organization of a software system****, the selection of* ***the structural elements and their interfaces*** *by which the system is composed,* ***together with their behavior*** *as specified in the collaborations among those elements, the* ***composition of these structural elements and behavioral elements into progressively larger subsystems****, and the architecture style that guides this organization — these elements and their interfaces, their collaborations, and their composition.*

Now, from the ANSI/IEEE Std 1471-2000 (the Open Group):

> *The* ***fundamental organization of a system****, embodied in its* ***components,*** *their* ***relationships*** *to each other and the environment, and the* ***principles governing its design and evolution****.*

In the Open Group Architecture Framework (TOGAF), *architecture* has two meanings depending on context (the Open Group):

1) A ***formal description of a system****, or a* ***detailed plan of the system at component level*** *to guide its implementation*

2) The ***structure of components****, their* ***inter-relationships****, and the* ***principles and guidelines governing their design and evolution over time***

Examining these definitions reveals many common keywords, which I've made bold in the various definitions. Important underlying currents are embodied by these keywords. But, these keywords lead to some important questions that must be answered to more fully understand architecture. What makes a decision architecturally significant? What are the elements of composition? How do we accommodate evolution of architecture? What does this have to do with modularity? As we delve into these questions, I want to start with a story on software architecture.

3.2 A Software Architecture Story

The story of software architecture reminds me of the following story (Hawking 1998):

A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise." The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down!"

—*A Brief History of Time* by Stephen Hawking

Software architecture is "turtles all the way down." How? This section discusses these ideas.

3.2.1 The Ivory Tower

Many of us can relate to the ivory tower. In dysfunctional organizations, architects and developers fail to communicate effectively. The result is a

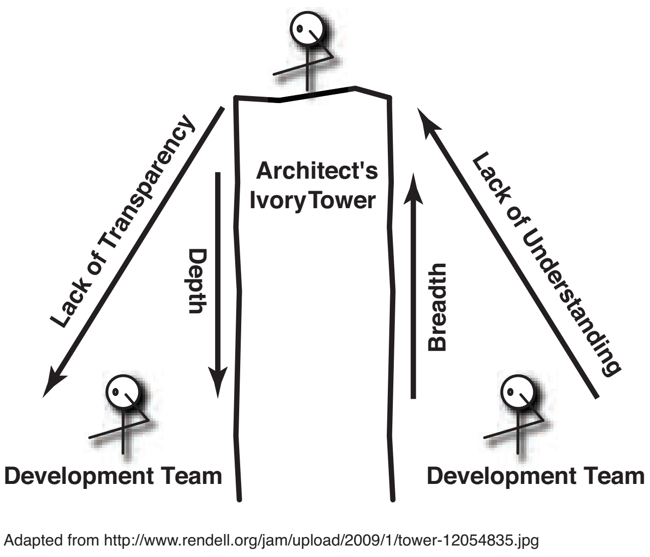


Figure 3.1 The ivory tower (the Open Group)

lack of transparency and a lack of understanding by both sides. As shown in Figure 3.1, architects bestow their wisdom upon developers who are unable to translate high-level concepts into concrete implementations. The failure often occurs (although I recognize there are other causes) because architecture is about breadth and development is about depth. Each group has disparate views of software architecture, and although both are warranted, there's a gap between these views. The architect might focus on applications and services, while the developer focuses on the code. Sadly, there is a lot in between that no one focuses on. This gap between breadth and depth contributes to ivory tower architecture.

3.2.2 Turtles and the Tower

Without question, the ivory tower is dysfunctional, and systems lacking architectural integrity are a symptom of ivory tower architecture. So, assuming good intent on the part of the architect and the developer, how can we bridge the gap between breadth and depth? How can we more effectively communicate? How do we increase understanding and transparency?

Let's revisit the definition of software architecture by exploring another definition. My favorite definition of software architecture was offered by Ralph Johnson in an article by Martin Fowler (2003). He states:

> *In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called "architecture." This understanding includes how the system is divided into components and how the components interact through interfaces. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers . . . Architecture is about the important stuff. Whatever that is.*

The key aspect of this definition that differentiates it from the earlier definitions in this chapter is that of "shared understanding," which implies that there is a social aspect to software architecture. We must have a shared understanding of how the system is divided into components and how they interact. Architecture isn't just some technical concept; it's also a social construct. Through this social aspect of architecture, we can break down the divide between architects and developers.

architecture all the way down, 69

To ensure shared understanding, we have to architect "all the way down." Architects cannot worry only about services, and developers cannot worry only about code. Each group must also focus on a huge middle ground, as illustrated in Figure 3.2.

Focusing exclusively on top-level abstractions is not enough. Emphasizing only code quality is not enough either. We must bridge the gap through other means, including module and package design. Often, when I speak at various conferences, I ask the audience to raise their hands if they devote effort to service design. Many hands raise. I also ask them to raise their hand if they spend time on class design and code quality. Again, many hands go up. But when I ask if they also devote effort to package and module design, only a small percentage leave their hands raised.

This is unfortunate, because module and package design are equally as important as service and class design. But somewhere along the way, with our emphasis on services and code quality, we've lost sight of what lies in between. Within each application or service awaits a rotting design, and atop even the most flexible code sits a suite of applications or services riddled with duplication and lack of understanding. A resilient package structure and corresponding software modules help bridge the divide between services and code. Modularity is an important intermediate

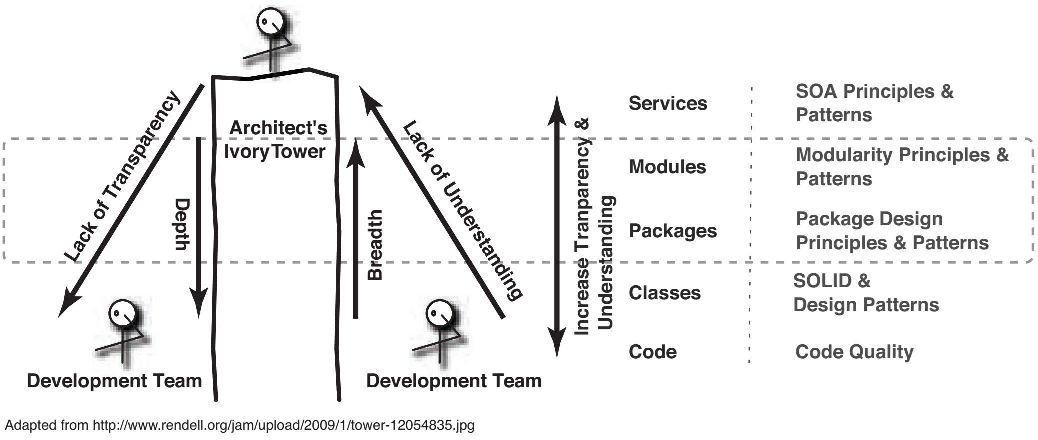

Architect's IvoryTower
Lack of Transparency
Depth
Breadth
Lack of Understanding
Development Team
Development Team
Increase Tranparency & Understanding
Services
SOA Principles & Patterns
Modules
Modularity Principles & Patterns
Packages
Package Design Principles & Patterns
Classes
SOLID & Design Patterns
Code
Code Quality
Adapted from http://www.rendell.org/jam/upload/2009/1/tower-12054835.jpg

Figure 3.2 Architecture all the way down

technology that helps us architect all the way down and is the conduit that fills the gap between breadth and depth.

We need to focus on modularity to ensure a consistent architecture story is told. It is the glue that binds. It's the piece that helps bridge low-level class design with higher-level service design. It's the piece that helps bring down the ivory tower, enhance communication, increase transparency, ensure understanding, and verify consistency at multiple levels. It is the piece that allows us to "architect all the way down" and allows us to realize the goal of architecture.

3.3 The Goal of Architecture

Modularity helps address the social aspect of software architecture, but it also helps us design more flexible software systems—that is, systems with resilient, adaptable, and maintainable architectures. Examining the earlier definitions of architecture leads us to the goal of architecture. The Johnson definition of architecture as quoted by Fowler makes it apparent that architecture is about the important stuff. In the following statement, Booch makes it clear that something is architecturally significant if it's difficult to change (2006):

All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

Based on these statements, it's fair to conclude that the goal of software architecture must be to eliminate the impact and cost of change, thereby eliminating architectural significance. We attempt to make something architecturally insignificant by creating flexible solutions that can be changed easily, as illustrated in Figure 3.3. But herein lies a paradox.

3.3.1 The Paradox

The idea behind eliminating architecture isn't new. In fact, Fowler mentions "getting rid of software architecture" in his article "Who Needs an Architect?" (2003). The way to eliminate architecture by minimizing the impact of cost and change is through flexibility. The more flexible the system, the more likely that the system can adapt and evolve as necessary. But herein lies the paradox, and a statement by Ralph Johnson presents and supports the idea (Fowler 2003):

. . . making everything easy to change makes the entire system very complex . . .

complexity, 46

As flexibility increases, so does the complexity. And complexity is the beast we are trying to tame because complex things are more difficult to deal with than simple things. It's a battle for which there is no clear path to victory, for sure. But, what if we were able to tame complexity while increasing flexibility, as illustrated in Figure 3.4? Let's explore the possibility of designing flexible software without increasing complexity. Is it even possible? In other words, how do we eliminate architecture?

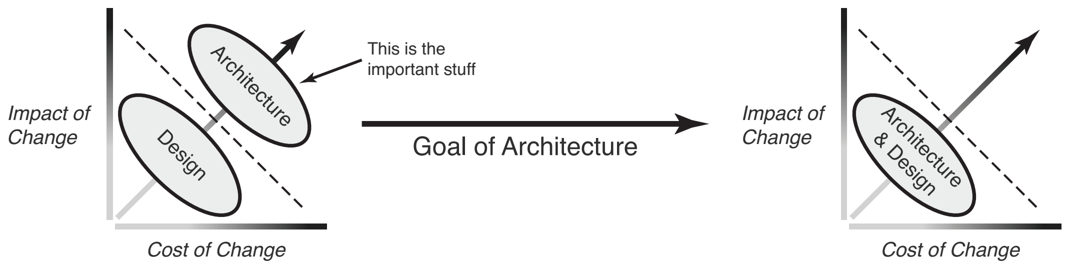


Figure 3.3 The goal of architecture

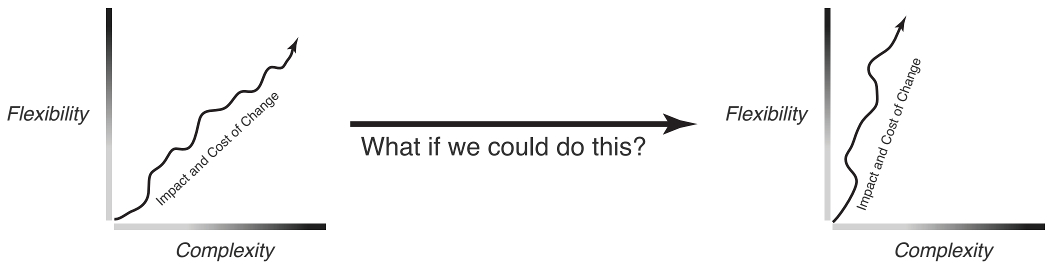


Figure 3.4 Maximizing flexibility, managing complexity

3.3.2 Eliminating Architecture

As the Johnson quote clearly points out, it's not feasible to design an infinitely flexible system. Therefore, it's imperative that we recognize where flexibility is necessary to reduce the impact and cost of change. The challenge is that we don't always know early in the project what might eventually change, so it's impossible to create a flexible solution to something we can't know about. This is the problem with Big Architecture Up Front (BAUF), and it's why we must make architectural decisions temporally. In other words, we should try to defer commitment to specific architectural decisions that would lock us to a specific solution until we have the requisite knowledge that will allow us to make the most informed decision.

It's also why we must take great care in insulating and isolating decisions we're unsure of and ensuring that these initial decisions are easy to change as answers to the unknown emerge. For this, modularity is a missing ingredient that helps minimize the impact and cost of change, and it's a motivating force behind why we should design software systems with a modular architecture. In the UML User Guide (page 163), Booch talks about "modeling the seams in a system." He states (1999):

> *Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side, as long as the components on both sides conform to the contract specified by that interface.*

Where Booch talks about components, we talk about modules. Where Booch talks about seams, we'll talk about joints. Modularity, combined with design patterns and SOLID principles, represents our best hope to

SOLID principles, 319

joints, 56

minimize the impact and cost of change, thereby eliminating the architectural significance of change.

3.4 Modularity: The Missing Ingredient

module definition, 17

Two of the key elements of the architectural definitions are component and composition. Yet there is no standard and agreed-upon definition of *component*[1] (reminding me of architecture, actually), and most use the term loosely to mean "a chunk of code." But, that doesn't work, and in the context of OSGi, it's clear that a module is a software component. Developing a system with an adaptive, flexible, and maintainable architecture requires modularity because we must be able to design a flexible system that allows us to make temporal decisions based on shifts that occur throughout development. Modularity has been a missing piece that allows us to more easily accommodate these shifts, as well as focus on specific areas of the system that demand the most flexibility, as illustrated in Figure 3.5. It's easier to change a design encapsulated within a module than it is to make a change to the design than spans several modules.

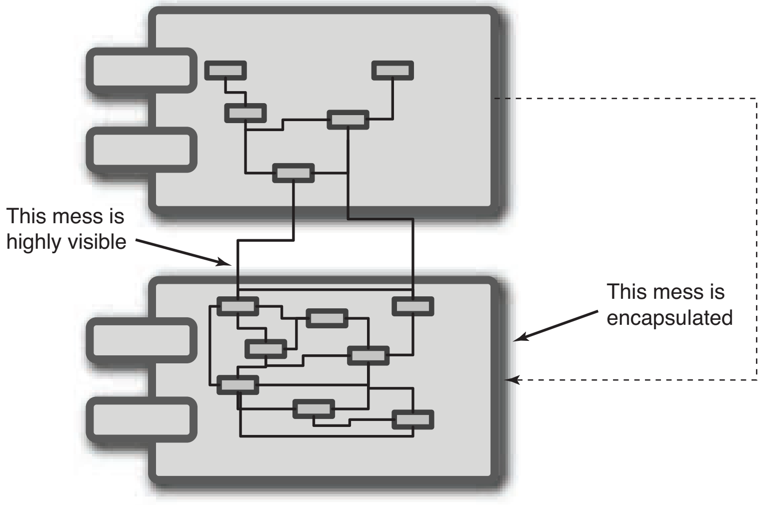


Figure 3.5 Encapsulating design

1. In his book *Component Software: Beyond Object-Oriented Programming*, Clemens Szyperski makes one of the few attempts I've seen to formally define the term *component*. He did a fine job, too.

3.4.1 Is It Really Encapsulated?

In standard Java, there is no way to enforce encapsulation of design details to a module because Java provides no way to define packages or classes that are module scope. As a result, classes in one module will always have access to the implementation details of another module. This is where a module framework, such as OSGi, shines because it allows you to forcefully encapsulate implementation details within a module through its explicit import package and export package manifest headers. Even public classes within a package cannot be accessed by another module unless the package is explicitly exported. The difference is subtle, although profound. We see several examples of this in the patterns throughout this book, and I point it out as it occurs. For now, let's explore a simple example.

modularizing without a runtime module, 26

3.4.1.1 Standard Java: No Encapsulation

Figure 3.6 illustrates a `Client` class that depends upon `Inter`, an interface, with `Impl` providing the implementation. The `Client` class is packaged in the `client.jar` module, and `Inter` and `Impl` are packaged in the `provider.jar` module. This is a good example of a modular system but demonstrates how we cannot encapsulate implementation details in standard Java because there is no way to prevent access to `Impl`. Classes

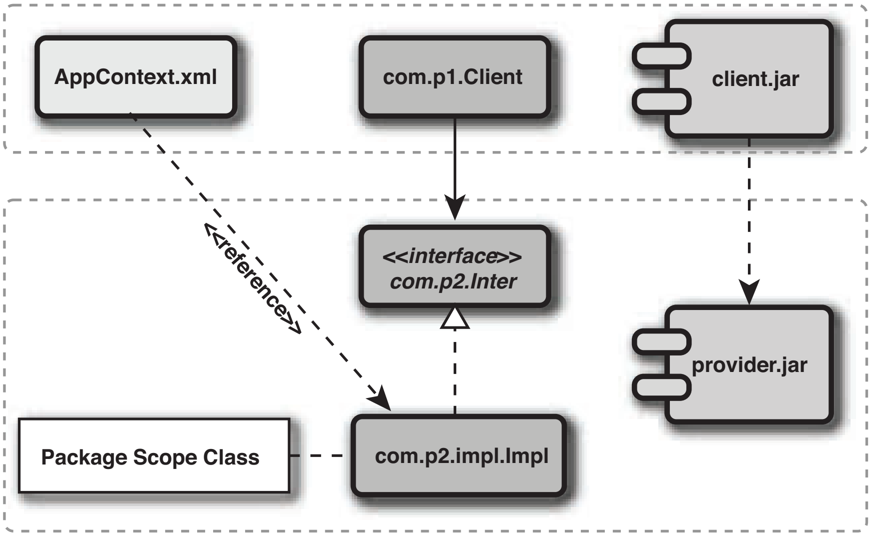


Figure 3.6 Standard Java can't encapsulate design details in a module.

outside of the `provider.jar` module can still reach the `Impl` class to instantiate and use it directly.

In fact, the `Impl` class is defined as a package scope class, as shown in Listing 3.1. However, the `AppContext.xml` Spring XML configuration file, which is deployed in the `client.jar` module, is still able to create the `Impl` instance at runtime and inject it into `Client`. The `AppContext.xml` and `Client` class are shown in Listing 3.2 and Listing 3.3, respectively. The key element is that the `AppContext.xml` is deployed in the `client.jar` module and the `Impl` class it creates is deployed in the `provider.jar` module. As shown in Listing 3.2, the `AppContext.xml` file deployed in the `client.jar` file violates encapsulation by referencing an implementation detail of the `provider.jar` module. Because the Spring configuration is a global configuration, the result is a violation of encapsulation.

Listing 3.1 Impl Class

```
package com.p2.impl;

import com.p2.*;

class Impl implements Inter {
      public void doIt() { . . . /* any implementation */ }
}
```

Listing 3.2 AppContext.xml Spring Configuration

```
<beans>
    <bean id="inter" class="com.p2.impl.Impl"/>
</beans>
```

Listing 3.3 Client Class

```
package com.p1;

import com.p2.*;
import org.springframework.context.*;
import org.springframework.context.support.*;

public class Client {
   public static void main(String args[]) {
      ApplicationContext appContext = new
      FileSystemXmlApplicationContext(
         "com/p1/AppContext.xml");
```

```
        Inter i = (Inter) appContext.getBean("inter");
        i.doIt();
    }
}
```

3.4.1.2 OSGi and Encapsulation

Now let's look at the same example using OSGi. Here, the `Impl` class in the `provider.jar` module is tightly encapsulated, and no class in any other module is able to see the `Impl` class. The `Impl` class and `Inter` interface remain the same as in the previous examples; no changes are required. Instead, we've taken the existing application and simply set it up to work with the OSGi framework, which enforces encapsulation of module implementation details and provides an intermodule communication mechanism. *OSGi, 273*

Figure 3.7 demonstrates the new structure. It's actually an example of the Abstract Modules pattern. Here, I separated the Spring XML *Abstract Modules pattern, 222*

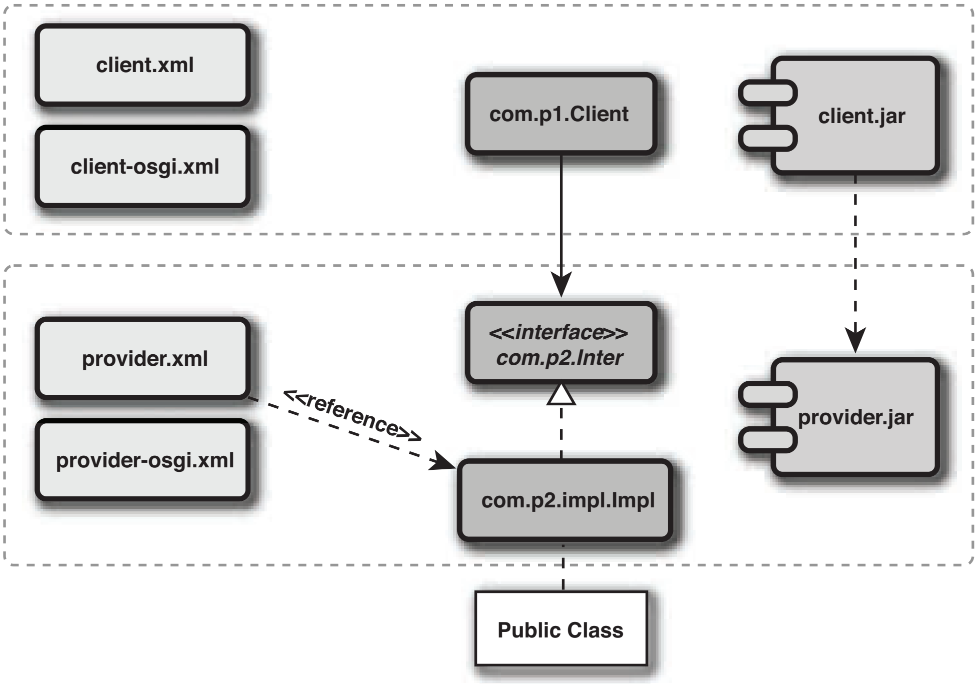


Figure 3.7 Encapsulating design with OSGi

configuration into four different files. I could have easily used only two configuration files, but I want to keep the standard Java and OSGi framework configurations separate for each module. The `provider.jar` module is responsible for the configuration itself and exposing its capabilities when it's installed. Before we describe the approach, here is a brief description of each configuration file:

- **client.xml**: Standard Spring configuration file that describes how the application should be launched by the OSGi framework
- **client-osgi.xml**: Spring configuration file that allows the Client class to consume an OSGi µService
- **provider.xml**: Spring configuration with the `provider.jar` module bean definition
- **provider-osgi.xml**: Spring configuration that exposes the bean definition in `provider.xml` as an OSGi µService

Before we look at how the two modules are wired together, let's look at the `provider.jar` module, which contains the `Inter` interface, `Impl` implementation, and two configuration files. Again, `Inter` and `Impl` remain the same as in the previous example, so let's look at the configuration files. The `provider.xml` file defines the standard Spring bean configuration and is what was previously shown in the `AppContext.xml` file in Figure 3.7. Listing 3.4 shows the `provider.xml` file. The key is that this configuration is deployed with the `provider.jar` module. Attempting to instantiate the `Impl` class outside of the `provider.jar` module will not work. Because OSGi enforces encapsulation, any attempt to reach the implementation details of a module will result in a runtime error, such as a `ClassNotFoundException`.

Listing 3.4 provider.xml Configuration File

```
<beans>
       <bean id="inter" class="com.p2.impl.Impl"/>
</beans>
```

How does OSGi prevent other classes from instantiating the `Impl` class directly? The `Manifest.mf` file included in the `provider.jar` module exposes classes only in the `com.p2` package, not the `com.p2.impl` package. So, the `Inter` interface registered as an OSGi µService is accessible by other modules but not by the `Impl` class. Listing 3.5 shows the section of the `Manifest.mf` illustrating the package export.

Listing 3.5 provider.xml Configuration File

```
Export-Package: com.p2
```

The `provider-osgi.xml` file is where things get very interesting, and it is where we expose the behavior of the `provider.jar` module as an OSGi µService that serves as the contract between the `Client` and `Impl` classes. The configuration for the `provider.jar` module lives within the `provider.jar` module, so no violation of encapsulation occurs.

Listing 3.6 shows the configuration. The name of the µService we are registering with the OSGi framework is called `interService`, and it references the `Impl` bean defined in Listing 3.4, exposing its behavior as type `Inter`. At this point, the `provider.jar` module has a `interService` OSGi µService that can be consumed by another module. This service is made available by the `provider.jar` module after it is installed and activated in the OSGi framework.

Listing 3.6 provider.xml Configuration File

```
<osgi:service id="interService" ref="inter"
      interface="com.p2.Inter"/>
```

Now, let's look at the `client.jar` module. The `client.xml` file configures the `Client` class. It effectively replaces the `main` method on the `Client` class in Listing 3.3 with the `run` method, and the OSGi framework instantiates the `Client` class, configures it with an `Inter` type, and invokes the `run` method. Listing 3.7 shows the `client.xml` file, and Listing 3.8 shows the `Client` class. This is the mechanism that initiates the process and replaces the `main` method in the `Client` class of the previous example.

Listing 3.7 Client.xml Configuration File

```
<beans>
      <bean name="client" class="com.p1.impl.Client"
      init-method="run">
                  <property name="inter"
                   ref="interService"/>
        </bean>
</beans>
```

Listing 3.8 The Client Class

```
package com.p1.impl;
import com.p2.*;
import com.p1.*;
public class Client {
      private Inter i;
      public void setInter(Inter i) {
            this.i = i;
      }

      public void run() throws Exception {
            i.doIt();
      }
}
```

The `Inter` type that is injected into the client class is done through the `client-osgi.xml` configuration file. Here, we specify that we want to use a µService of type `Inter`, as shown in Listing 3.9.

Listing 3.9 Client.xml Configuration File

```
<osgi:reference id="interService"
 interface="com.p2.Inter"/>
```

The `Manifest.mf` file for the `client.jar` module imports the `com.p2` packages, which gives it access to the `Inter` µService. Listing 3.10 shows the section of `Manifest.mf` showing the package imports and exports for the `client.jar` module.

Listing 3.10 Client.xml Configuration File

```
Import-Package: com.p2
```

Independent Deployment pattern, 178

This simple example has several interesting design aspects.[2] The `provider.jar` module is independently deployable. It has no dependencies on any other module, and it exposes its set of behaviors as a µService. No other module in the system needs to know these details.

2. Although this example builds upon the OSGi Blueprint Specification, some of you may not be huge fans of XML. If that's the case, Peter Kriens has an implementation that uses OSGi Declarative Services. The sample can be found at http://bit.ly/OSGiExamples in the aQute.poma.basic directory.

The design could have also been made even more flexible by packaging the `Impl` class and `Inter` interface in separate modules. By separating the interface from the implementation, we bring a great deal of flexibility to the system, especially with OSGi managing our modules.

Separate Abstractions pattern, 237

At first glance, it might also appear to contradict the External Configuration pattern. When defining the external configuration for a module, we still want to ensure implementation details are encapsulated. External configuration is more about allowing clients to configure a module to its environmental context and not about exposing implementation details of the module.

External Configuration pattern, 200

The key takeaway from this simple demonstration is that the classes in the `provider.jar` module are tightly encapsulated because the OSGi framework enforces type visibility. We expose only the public classes in the packages that a module exports, and the µService is the mechanism that allows modules to communicate in a very flexible manner. The µService spans the joints of the system, and because OSGi is dynamic, so too are the dependencies on µServices. Implementations of the µService can come and go at runtime, and the system can bind to new instances as they appear.

joints, 56

Again, we'll see several more examples of this throughout the remainder of the discussion. Even though you can't enforce encapsulation of module implementation using standard Java, it's still imperative to begin designing more modular software systems. As we'll see, by applying several of the techniques we discuss in this book, we put ourselves in an excellent position to take advantage of a runtime module system.

3.5 Answering Our Questions

Earlier, this chapter posed the following questions after introducing the three definitions of software architecture. Through explanation, we answered each question. But to be clear, let's offer concise answers:

What makes a decision architecturally significant? A decision is architecturally significant if the impact and cost of change is significant.

What are the elements of composition? The elements of composition include classes, modules, and services.

How do we accommodate evolution of architecture? Evolution is realized by designing flexible solutions that can adapt to change. But flexibility breeds complexity, and we must be careful to build flexibility in the right areas of the system.

3.6 Conclusion

The goal of architecture is to minimize the impact and cost of change. Modularity helps us realize this goal by filling in a gap that exists between top-level architectural constructs and lower-level code. Modularity is the important intermediate that helps increase architectural agility. It fills a gap that exists between architects and developers. It allows us to create a software architecture that can accommodate shifts. Modularity helps us architect all the way down.

3.7 References

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

The Open Group. *The Open Group Architecture Framework*. www.opengroup.org/architecture/togaf8-doc/arch/chap01.html

Hawking, Stephen. 1998. *A Brief History of Time*. Bantam.

Fowler, Martin. 2003. "Who Needs an Architect?" IEEE Software.

Booch, Grady. 2006. *On Design*. www.handbookofsoftwarearchitecture.com/index.jsp?page=Blog&part=All

Index