Large-Scale C++
Volume I

Process and Architecture

John Lakos

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Large-Scale C++

This page intentionally left blank

Large-Scale C++

Volume I

Process and Architecture

John Lakos

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019948467

Cover image: MBoe/Shutterstock

ISBN-13: 978-0-201-71706-8
ISBN-10: 0-201-71706-9

ScoutAutomatedPrintLine

To my wife, Elyse, with whom the universe rewarded me,
and five wonderful children:
Sarah
Michele
Gabriella
Lindsey
Andrew

Contents

Preface

When I wrote my first book, *Large-Scale C++ Software Design* (**lakos96**), my publisher wanted me to consider calling it *Large-Scale C++ Software* Development. I was fairly confident that I was qualified to talk about design, but the topic of *development* incorporated far more scope than I was prepared to address at that time.

Design, as I see it, is a static property of software, most often associated with an individual application or library, and is only one of many disciplines needed to create successful software. *Development*, on the other hand, is dynamic, involving people, processes, and workflows. Because development is ongoing, it typically spans the efforts attributed to many applications and projects. In its most general sense, development includes the design, implementation, testing, deployment, and maintenance of a series of products over an extended period. In short, software development is what we *do*.

In the more than two decades following *Large-Scale C++ Software Design*, I consistently applied the same fundamental design techniques introduced there (and elucidated here), both as a consultant and trainer and in my full-time work. I have learned what it means to assemble, mentor, and manage large development teams, to interact effectively with clients and peers, and to help shape corporate software engineering culture on an enterprise scale. Only in the wake of this additional experience do I feel I am able to do justice to the much more expansive (and ambitious) topic of large-scale software *development*.

A key principle — one that helps form the foundation of this multivolume book — is the profound importance of organization in software. Real-world software is intrinsically complex; however, a great deal of software is needlessly complicated, due in large part to a lack of basic organization — both in the way in which it is developed and in the final form that it takes. This book is first and foremost about what constitutes well-organized software, and also about the processes, methods, techniques, and tools needed to realize and maintain it.

Secondly, I have come to appreciate that not all software is or should be created with the same degree of polish. The value of real-world application software is often measured by how fast code gets to market. The goals of the software engineers apportioned to application development projects will naturally have a different focus and time frame than those slated to the long-term task of developing reliable and reusable software infrastructure. Fortunately, all of the techniques discussed in this book pertain to both application and library software — the difference being the extent to and rigor with which the various design, documentation, and testing techniques are applied.

One thing that has not changed and that has been proven repeatedly is that all real-world software benefits from *physical design*. That is, the way in which our logical content is factored and partitioned within files and libraries will govern our ability to identify, develop, test, maintain, and reuse the software we create. In fact, the architecture that results from thoughtful physical design at every level of aggregation continues to demonstrate its effectiveness in industry every day. Ensuring sound physical design, therefore, remains the first pillar of our methodology, and a central organizing principle that runs throughout this three-volume book — a book that both captures and expands upon my original work on this subject.

The second pillar of our methodology, nascent in *Large-Scale C++ Software Design*, involves essential aspects of *logical design* beyond simple syntactic rendering (e.g., *value semantics*). Since C++98, there has been explosive growth in the use of templates, generic programming, and the Standard Template Library (STL). Although templates are unquestionably valuable, their aggressive use can impede interoperability in software, especially when generic programming is not the right answer. At the same time, our focus on enterprise-scale development and our desire to maximize *hierarchical* reuse (e.g., of memory allocators) compels reexamination of the proper use of more mature language constructs, such as (public) inheritance.

Maintainable software demands a well-designed interface (for the compiler), a concise yet comprehensive contract (for people), and the most effective implementation techniques available (for efficiency). Addressing these along with other important *logical design* issues, as well as providing advice on implementation, documentation, and rendering, rounds out the second part of this comprehensive work.

Verification, including testing and static analysis, is a critically important aspect of software development that was all but absent in *Large-Scale C++ Software Design* and limited to *testability* only. Since the initial publication of that book, teachable testing strategies, such as Test-Driven Development (TDD), have helped make testing more fashionable today than it was in the 1990s or even in the early 2000s. Separately, with the start of the millennium, more and more companies have been realizing that thorough unit testing *is* cost-effective (or at least less expensive than not testing). Yet what it means to test continues to be a black art, and all too often "unit testing" remains little more than a checkbox in one's prescribed SOP (Standard Operating Procedure).

As the third pillar of our complete treatment of component-based software development, we address the discipline of creating effective unit tests, which naturally double as regression tests. We begin by delineating the underlying concept of what it means to test, followed by how to (1) select test input systematically, (2) design, implement, and render thorough test cases readably, and (3) optimally organize component-level test drivers. In particular, we discuss deliberately ordering test cases so that primitive functionality, once tested, can be leveraged to test other functionality within the same component.

Much thought was given to choosing a programming language to best express the ideas corresponding to these three pillars. C++ is inherently a compiled language, admitting both preprocessing and separate translation units, which is essential to fully addressing all of the important concepts pertaining to the dimension of software engineering that we call *physical design*. Since its introduction in the 1980s, C++ has evolved into a language that supports multiple programming paradigms (e.g., functional, procedural, object-oriented, generic), which invites discussion of a wide range of important *logical design* issues (e.g., involving templates, pointers, memory management, and maximally efficient spatial and/or runtime performance), not all of which are enabled by other languages.

Since *Large-Scale C++ Software Design* was published, C++ has been standardized and extended many times and several other new and popular languages have emerged.[1] Still, for both practical and pedagogical reasons, the subset of modern C++ that is C++98 remains the language of choice for presenting the software engineering principles described here. Anyone

[1] In fact, much of what is presented here applies analogously to other languages (e.g., Java, C#) that support separate compilation units.

who knows a more modern dialect of C++ knows C++98 but not necessarily vice versa. All of the theory and practice upon which the advice in this book was fashioned is independent of the particular subset of the C++ language to which a given compiler conforms. Superficially retrofitting code snippets (used from the inception of this book) with the latest available C++ syntax — just because we're "supposed to" — would detract from the true purpose of this book and impede access to those not familiar with modern C++.[2] In those cases where we have determined that a later version of C++ could afford a clear win (e.g., by expressing an idea significantly better), we will point them out (typically as a footnote).

This methodology, which has been successfully practiced for decades, has been independently corroborated by many important literary references. Unfortunately, some of these references (e.g., **stroustrup00**) have since been superseded by later editions that, due to covering new language features and to space limitations, no longer provide this (sorely needed) design guidance. We unapologetically reference them anyway, often reproducing the relevant bits here for the reader's convenience.

Taken as a whole, this three-volume work is an engineering reference for software developers and is segmented into three distinct, physically separate volumes, describing in detail, from a developer's perspective, *all* essential technical[3] aspects of this proven approach to creating an organized, integrated, scalable software development environment that is capable of supporting an entire enterprise and whose effectiveness only improves with time.

Audience

This multivolume book is written explicitly for practicing C++ software professionals. The sequence of material presented in each successive volume corresponds roughly to the order in which developers will encounter the various topics during the normal design-implementation-test cycle. This material, while appropriate for even the largest software development organizations, applies also to more modest development efforts.

---

[2] Even if we had chosen to use the latest C++ constructs, we assert that the difference would not be nearly as significant as some might assume.

[3] This book does not, however, address some of the softer skills (e.g., requirements gathering) often associated with full lifecycle development but does touch on aspects of project management specific to our development methodology.

Application developers will find the organizational techniques in this book useful, especially on larger projects. It is our contention that the rigorous approach presented here will recoup its costs within the lifetime of even a single substantial real-world application.

Library developers will find the strategies in this book invaluable for organizing their software in ways that maximize reuse. In particular, packaging software as an acyclic hierarchy of fine-grained physical *components* enables a level of quality, reliability, and maintainability that to our knowledge cannot be achieved otherwise.

Engineering managers will find that throttling the degree to which this suite of techniques is applied will give them the control they need to make optimal schedule/product/cost trade-offs. In the long term, consistent use of these practices will lead to a repository of *hierarchically reusable* software that, in turn, will enable new applications to be developed faster, better, and cheaper than they could ever have been otherwise.

Roadmap

Volume I (the volume you're currently reading) begins this book with our domain-independent software process and architecture (i.e., how *all* software should be created, rendered, and organized, no matter what it is supposed to do) and culminates in what we consider the state-of-the-art in physical design strategies.

Volume II (forthcoming) continues this multivolume book to include large-scale logical design, effective component-level interfaces and contracts, and highly optimized, high-performance implementation.

Volume III (forthcoming) completes this book to include verification (especially unit testing) that maximizes quality and leads to the cost-effective, fine-grained, *hierarchical* reuse of an ever-growing repository of *Software Capital*.[4]

The entire multivolume book is intended to be read front-to-back (initially) and to serve as a permanent reference (thereafter). A lot of the material presented will be new to many readers. We have, therefore, deliberately placed much of the more difficult, detailed, or in some sense "optional" material toward the end of a given chapter (or section) to allow the reader to skim (or skip) it, thereby facilitating an easier first reading.

[4] See section 0.9.

We have also made every effort to cross-reference material across all three volumes and to provide an effective index to facilitate referential access to specific information. The material naturally divides into three parts: (I) Process and Architecture, (II) Design and Implementation, and (III) Verification and Testing, which (not coincidentally) correspond to the three volumes.

Volume I: Process and Architecture

Chapter 0, "Motivation," provides the initial engineering and economic incentives for implementing our scalable development process, which facilitates hierarchical reuse and thereby simultaneously achieves shorter time to market, higher quality, and lower overall cost. This chapter also discusses the essential dichotomy between infrastructure and application development and shows how an enterprise can leverage these differences to improve productivity.

Chapter 1, "Compilers, Linkers, and Components," introduces the *component* as the fundamental atomic unit of logical and physical design. This chapter also provides the basic low-level background material involving compilers and linkers needed to absorb the subtleties of the main text, building toward the definition and essential properties of components and physical dependency. Although nominally background material, the reader is advised to review it carefully because it will be assumed knowledge throughout this book and it presents important vocabulary, some of which might not *yet* be in mainstream use.

Chapter 2, "Packaging and Design Rules," presents how we organize and package our component-based software in a uniform (domain-independent) manner. This chapter also provides the fundamental design rules that govern how we develop modular software hierarchically in terms of components, packages, and package groups.

Chapter 3, **"Physical Design and Factoring,"** introduces important physical design concepts necessary for creating sound software systems. This chapter discusses proven strategies for designing large systems in terms of smaller, more granular subsystems. We will see how to partition and aggregate logical content so as to avoid cyclic, excessive, and otherwise undesirable (or unnecessary) physical dependencies. In particular, we will observe how to avoid the heaviness of conventional *layered* architectures by employing more *lateral* ones, understand how to reduce compile-time coupling at an architectural level, and learn — by example — how to design effectively using components.

Volume II: Design and Implementation (Forthcoming)

Chapter 4, "Logical Interoperability and Testability," discusses central, logical design concepts, such as *value semantics* and *vocabulary types*, that are needed to achieve interoperability and testability, which, in turn, are key to enabling successful reuse. It is in this chapter that we first characterize the various common class categories that we will casually refer to by name, thus establishing a context in which to more efficiently communicate well-understood families of behavior. Later sections in this chapter address how judicious use of templates, proper use of inheritance, and our fiercely modular approach to resource management — e.g., local ("arena") memory allocators — further achieve interoperability and testability.

Chapter 5, "Interfaces and Contracts," addresses the details of shaping the interfaces of the components, classes, and functions that form the building blocks of all of the software we develop. In this chapter we discuss the importance of providing well-defined contracts that clearly delineate, in addition to any object invariants, both what is *essential* and what is *undefined* behavior (e.g., resulting from *narrow* contracts). Historically controversial topics such as *defensive programming* and the explicit use of exceptions within contracts are addressed along with other notions, such as the critical distinction between *contract checking* and *input validation*. After attending to backward compatibility (e.g., physical substitutability), we address various facets of good contracts, including stability, `const`-correctness, reusability, validity, and appropriateness.

Chapter 6, "Implementation and Rendering," covers the many details needed to manufacture high-quality components. The first part of this chapter addresses some important considerations from the perspective of a single component's implementation; the latter part provides substantial guidance on minute aspects of consistency that include function naming, parameter ordering, argument passing, and the proper placement of operators. Toward the end of this chapter we explain — at some length — our rigorous approach to embedded component-level, class-level, and especially function-level documentation, culminating in a developer's final "checklist" to help ensure that all pertinent details have been addressed.

Volume III: Verification and Testing (Forthcoming)

Chapter 7, "Component-Level Testing," introduces the fundamentals of testing: what it means to test something, and how that goal is best achieved. In this (uncharacteristically) concise chapter, we briefly present and contrast some classical approaches to testing (less-well-factored) software, and we then go on to demonstrate the overwhelming benefit of insisting that each component have a single dedicated (i.e., standalone) test driver.

Chapter 8, "Test-Data Selection Methods," presents a detailed treatment of how to choose the input data necessary to write tests that are thorough yet run in near minimal time. Both classical and novel approaches are described. Of particular interest is *depth-ordered enumeration,* an original, systematic method for enumerating, in order of importance, increasingly complex tests for value-semantic container types. Since its initial debut in 1997, the sphere of applicability for this surprisingly powerful test-data selection method has grown dramatically.

Chapter 9, "Test-Case Implementation Techniques," explores different ways in which previously identified sampling data can be delivered to the functionality under test, and the results observed, in order to implement a valid test suite. Along the way, we will introduce useful concepts and machinery (e.g., *generator functions*) that will aid in our testing efforts. Complementary test-case implementation techniques (e.g., *orthogonal perturbation*), augmenting the basic ones (e.g., the *table-driven* technique), round out this chapter.

Chapter 10, "Test-Driver Organization," illustrates the basic organization and layout of our component-level test driver programs. This chapter shows how to order test cases optimally so that the more primitive methods (e.g., *primary manipulators* and *basic accessors*) are tested first and then subsequently relied upon to test other, less basic functionality defined within the same component. The chapter concludes by addressing the various major categories of classes discussed in Chapter 4; for each category, we provide a recommended test-case ordering along with corresponding test-case implementation techniques (Chapter 9) and test-data selection methods (Chapter 8) based on fundamental principles (Chapter 7).

Acknowledgments

Where do I start? Chapter 7, the one first written (c. 1999), of this multivolume book was the result of many late nights spent after work at Bear Stearns collaborating with Shawn Edwards, an awesome technologist (and dear friend). In December of 2001, I joined Bloomberg, and Shawn joined me there shortly thereafter; we have worked together closely ever since. Shawn assumed the role of CTO at Bloomberg LP in 2010.

After becoming hopelessly blocked trying to explain low-level technical details in Chapter 1 (c. 2002), I turned to another awesome technologist (and dear friend), Sumit Kumar, who actively coached me through it and even rewrote parts of it himself. Sumit — who might be the best programmer I've ever met — continues to work with me, providing both constructive feedback and moral support.

When I became overwhelmed by the sheer magnitude of what I was attempting to do (c. 2005), I found myself talking over the phone for nearly six hours to yet another awesome technologist (and dear friend), Vladimir Kliatchko, who walked me through my entire table of contents — section by section — which has remained essentially unchanged ever since. In 2012, Vlad assumed the role of Global Head of Engineering at Bloomberg and, in 2018, was appointed to Bloomberg's Management Committee.

John Wait, the Addison-Wesley acquisitions editor principally responsible for enabling my first book, wisely recommended (c. 2006) that I have a structural editor, versed in both writing and computer science, review my new manuscript for macroscopic organizational improvements. After review, however, this editor fairly determined that no reliable, practicable advice with respect to restructuring my copious writing would be forthcoming.

Eventually (c. 2010), yet another awesome technologist, Jeffrey Olkin, joined Bloomberg. A few months later, I was reviewing a software specification from another group. The documentation was good but not stellar — at least not until about the tenth page, after which it was perfect! I walked over to the titular author and asked what happened. He told me that Jeffrey had taken over and finished the document. Long story short, I soon after asked Jeffrey to act as my structural editor, and he agreed. In the years since, Jeffrey reviewed and helped me to rework every last word of this first volume. I simply cannot overstate the organizational, writing, and engineering contributions Jeffrey has made to this book so far. And, yes, Jeffrey too has become a dear friend.

There are at least five other technically expert reviewers that read this entire manuscript as it was being readied for publication and provided amazing feedback: JC van Winkel, David Sankel, Josh Berne, Steven Breitstein (who meticulously reviewed each of my figures after their translation from ASCII art), and Clay Wilson (a.k.a. "The Closer," for the exceptional quality of his code reviews). Each of these five senior technologists (the first three being members of the C++ Standards Committee; the last four being current and former employees of Bloomberg) has, in his own respectively unique way, made this book substantially more valuable as a result of his extensive, thoughtful, thorough, and detailed feedback.

There are many other folks who have contributed to this book from its inception, and some even before that. Professor Chris Van Wyc (Drew University), a principal reviewer of my first book, provided valuable organizational feedback on a nascent draft of this volume. Tom Marshall (who also worked with me at Bear Stearns) and Peter Wainwright have worked with me at Bloomberg since 2002 and 2003, respectively. Tom went on to become the head of the architecture office at Bloomberg, and Peter, the head of Bloomberg's SI Build team. Each of them has amassed a tremendous amount of practical knowledge relating to metadata (and the tools that use it) and were kind enough to have co-authored an entire section on that topic (see section 2.16).

Early in my tenure at Bloomberg (c. 2004), my burgeoning BDE[5] team was suffering from its own success and I needed reinforcements. At the time, we had just hired several more-senior folks (myself included) and there was no senior headcount allotted. I went with Shawn to the then head of engineering, Ken Gartner, and literally begged him to open five "junior" positions. Somehow he agreed, and within no time, all of the positions were filled by five truly outstanding candidates — David Rubin, Rohan Bhindwale, Shezan Baig, Ujjwal Bhoota, and Guillaume Morin — four by the same recruiter, Amy Resnik, who I've known since 1991 (her boss, Steven Markmen, placed me at Mentor Graphics in 1986). Every one of these journeyman engineers went on to contribute massively to Bloomberg's software infrastructure, two of them rising to the level of team lead, and one to manager; in fact, it was Guillaume who, having only 1.5 years of work experience, implemented (as his very first assignment) the "designing with components" example that runs throughout section 3.12.

In June 2009, I recall sitting in the conference hotel for the C++ Standard Committee meeting in Frankfurt, Germany, having a "drink" (soda) with Alisdair Meredith — soon to be the library working group (LWG) chair (2010-2015) — when I got a call from a recruiter (Amy Resnik, again), who said she had found the perfect candidate to replace (another dear friend) Pablo Halpern on Bloomberg's BDE team (2003-2008) as our resident authority on the C++ Standard. You guessed it: Alisdair Meredith joined Bloomberg and (soon after) my BDE team in 2009, and ever since has been my definitive authority (and trusted friend) on what *is* in C++. Just prior to publication, Alisdair thoroughly reviewed the first three sections of Chapter 1 to make *absolutely sure* that I got it right.

Many others at Bloomberg have contributed to the knowledge captured in this book: Steve Downey was the initial architect of the **ball** logger, one of the first major subsystems developed at Bloomberg using our component-based methodology; Jeff Mendelson, in addition to providing many excellent technical reviews for this book, early on produced much of our modern date-math infrastructure; Mike Giroux (formerly of Bear Stearns) has historically been my able toolsmith and has crafted numerous custom Perl scripts that I have used throughout the years to keep my ASCII art in sync with ASCII text; Hyman Rosen, in addition to providing several

[5] BDE is an acronym for BDE Development Environment. This acronym is modeled after ODE (Our Development Environment) coined by Edward ("Ned") Horn at Bear Stearns in early 1997. The 'B' in BDE originally stood for "Bloomberg" (a common prefix for new subsystems and suborganizations of the day, e.g., *bpipe, bval, blaw*) and later also for "Basic," depending on the context (e.g., whether it was work or book related). Like ODE, BDE initially referred simultaneously to the lowest-level library package group (see section 2.9) in our Software-Capital repository (see section 0.5) along with the development team that maintained it. The term *BDE* has long since taken on a life of its own and is now used as a moniker to identify many different kinds of entities: *BDE* Group, *BDE* methodology, *BDE* libraries, *BDE* tools, *BDE* open-source repository, and so on; hence, the *recursive* acronym: BDE Development Environment.

unattributed passages in this book, has produced (over a five-year span) a prodigious (clang-based) static-analysis tool, **`bde_verify`**,[6] that is used throughout Bloomberg Engineering to ensure that conforming component-based software adheres to the design rules, coding standards, guidelines, and principles advocated throughout this book.

I would be remiss if I didn't give a shout-out to all of the *current* members of Bloomberg's BDE team, which I founded back in 2001, and, as of April 2019, is now managed by Mike Verschell along with Jeff Mendelsohn: Josh Berne, Steven Breitstein, Nathan Burgers, Bill Chapman, Attila Feher, Mike Giroux, Rostislav Khlebnikov, Alisdair Meredith, Hyman Rosen, and Oleg Subbotin. Most, if not all, of these folks have reviewed parts of the book, contributed code examples, helped me to render complex graphs or write custom tools, or otherwise in some less tangible way enhanced the value of this work.

Needless to say, without the unwavering support of Bloomberg's management team from Vlad and Shawn on down, this book would not have happened. My thanks to Andrei Basov (my current boss) and Wayne Barlow (my previous boss) — both also formerly of Bear Stearns — and especially to Adam Wolf, Head of Software Infrastructure at Bloomberg, for not just allowing but encouraging *and enabling* me (after some twenty-odd years) to finally realize this first volume.

And, of course, none of this would have been possible had Bjarne Stroustrup somehow decided to do anything other than make the unparalleled success of C++ his lifework. I have known Bjarne since he gave a talk at Mentor Graphics back in the early 1990s. (But he didn't know me then.) I had just methodically read *The Annotated C++ Reference Manual* (**ellis90**) and thoroughly annotated it (in four different highlighter colors) myself. After his talk, I asked Bjarne to sign my well-worn copy of the *ARM*. Decades later, I reminded him that it was I who had asked him to sign that disheveled, multicolored book of his; he recalled that, at least. Since becoming a regular attendee of the C++ Standards Committee meetings in 2006, Bjarne and I have worked closely together — e.g., to bring a better version of BDE's (library-based) **bsls_assert** contract-assertions facility, used at Bloomberg since 2004, into the language itself (see Volume II, section 6.8). Bjarne has spoken at Bloomberg multiple times at my behest. He reviewed and provided feedback on an early version of the preface of this book (minus these acknowledgments) and has also supplied historical data for footnotes. The sage software engineering wisdom from his special edition (third edition) of *The C++ Programming Language* (**stroustrup00**) is quoted liberally throughout this volume. Without his inspiration and encouragement, my professional life would be a far cry from what it is today.

[6] https://github.com/bloomberg/bde_verify

Finally, I would like to thank all of the many generations of folks at Pearson who have waited patiently for me throughout the years to get this book done. The initial draft of the manuscript was originally due in September 2001, and my final deadline for this first volume was at the end of September 2019. (It appears I'm a skosh late.) That said, I would like to recognize Debbie Lafferty, my first editor who then (in the early 2000s) passed the torch to Peter Gordon and Kim Spenceley (née Boedigheimer) with whom I worked closely for over a decade. When Peter retired in 2016, I began working with my current editor, Greg Doench.

Although Peter was a tough act to follow, Greg rose to the challenge and has been there for me throughout (and helped me more than he probably knows). Greg then introduced me to Julie Nahil, who worked directly with me on readying this book for production. In 2017, I reconnected with my lifelong friend and now wife, Elyse, who tirelessly tracked down copious references and proofread key passages (like this one). By late 2018, it became clear that the amount of work required to produce this book would exceed what anyone had anticipated, and so Pearson retained Lori Hughes to work with me, in what turned out to be a nearly full-time capacity for the better part of 2019. I cannot say enough about the professionalism, fortitude, and raw effort put forth by Lori in striving to make this book a reality in calendar year 2019. I want to thank Lori, Julie, and Greg, and also Peter, Kim, and Debbie, for all their sustained support and encouragement over so many, many years. And this is but the first of three volumes, OMG!

The list of people that have contributed directly and/or substantially to this work is dauntingly large, and I have no doubt that, despite my efforts to the contrary, many will go unrecognized here. Know that I realize this book is the result of my life's experiences, and for each of you that have in some way contributed, please accept my heartfelt thanks and appreciation for being a part of it.

2.1 The Big Picture

The way in which software is organized governs the degree to which we can leverage that software to solve current and new business problems quickly and effectively. By design, much of the code that we write for use by applications will reside in sharable libraries and not directly in any one application. Our goal, therefore, is to provide some top-level organizational structure — such as the one illustrated in Figure 2-1 — that allows us to partition our software into discrete physical units so as to facilitate finding, understanding, and potentially reusing available software solutions.

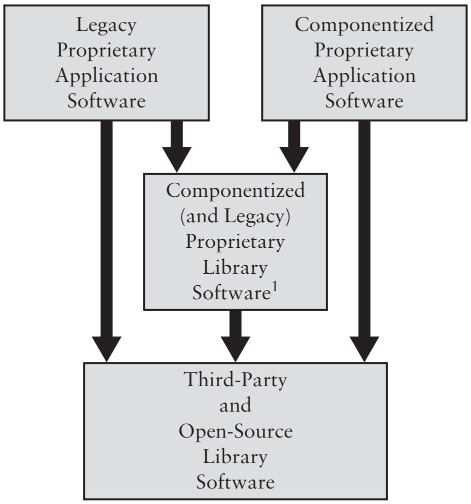


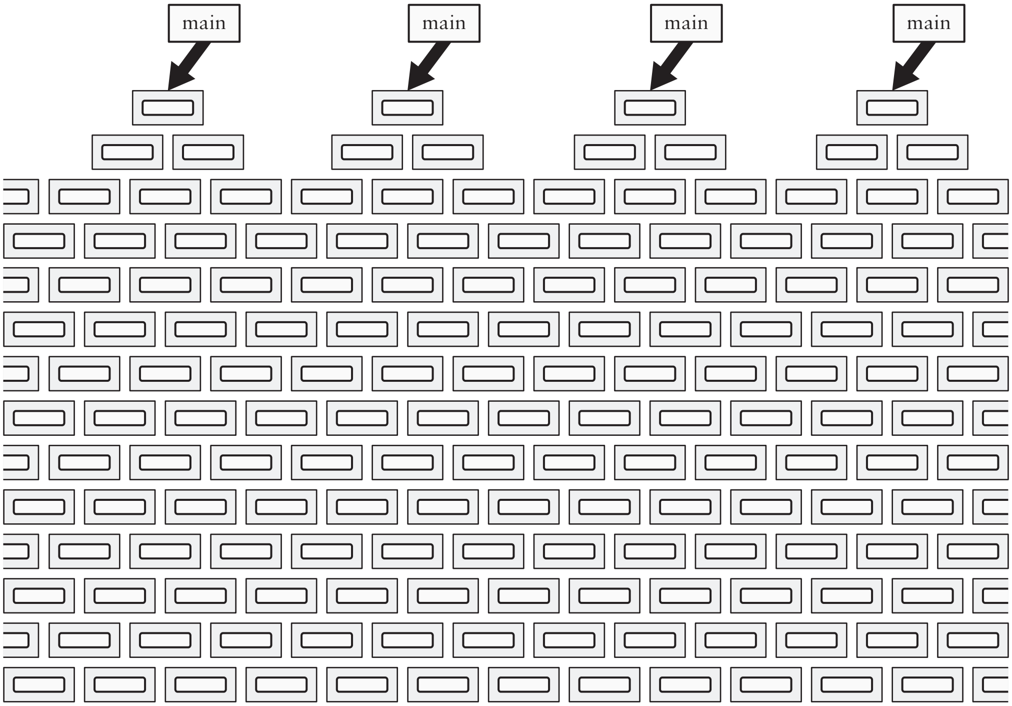
Figure 2-1: Enterprise-level view of software organization

As Chapters 0 and 1 describe, most of what we do with respect to creating new library and application software involves components as the atomic units of design. But components alone, as depicted in Figure 2-2a, are too small to be effective in managing and maintaining software on a large scale. We will therefore want to aggregate logically related components having similar physical dependencies into a larger physical entity that we refer to as a *package*, which can be treated more effectively as a unit. These larger logically and physically cohesive

---

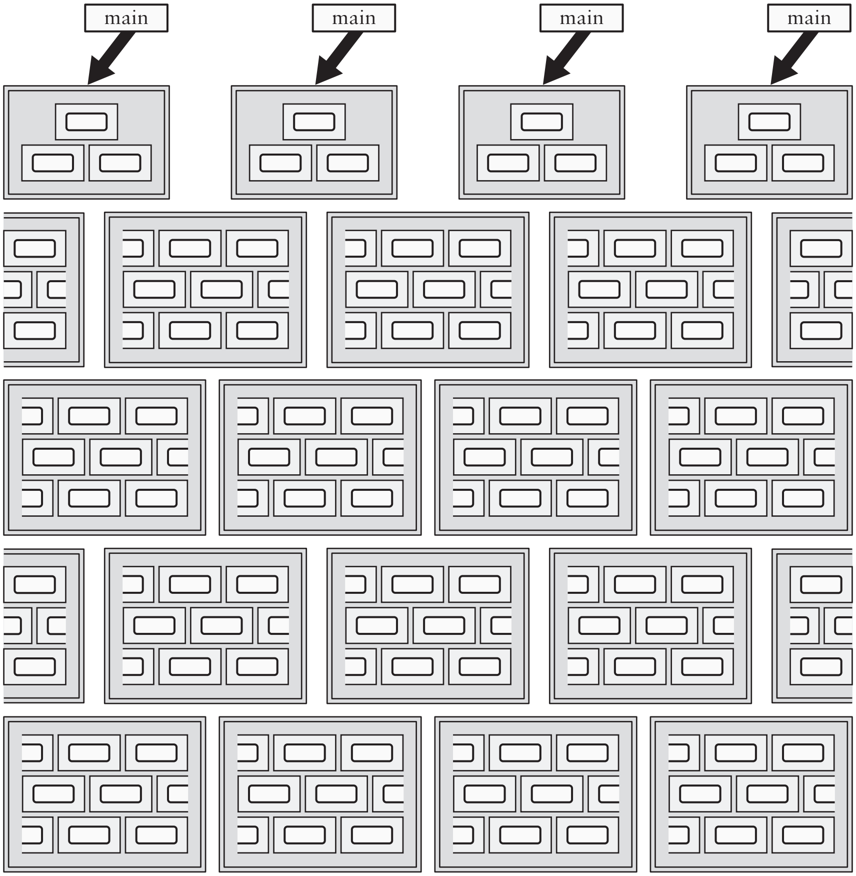
[1] Open-source code that has been augmented (or forked) to achieve some particular purpose would also fall into this category (e.g., third-party software adapted to use our (polymorphic) memory-allocator model — see Volume II, section 4.10).

entities can then, in turn, be further aggregated into a yet larger body of software, which we call a *package group*, comprising packages having similar physical dependencies[2] that, taken as a whole, are suitable for independent release, as illustrated in Figure 2-2b.



(a) System consisting of individual components

[2] Note that, while the packages within a group are themselves necessarily internally logically cohesive, such need not be the case for a package group as a whole (see sections 2.8 and 2.9, respectively).



(b) System consisting of pre-aggregated components

Figure 2-2: Individual components do not scale up.

In addition, some of the software that we might need to use could be organized quite differently. For example, we may want to take advantage of certain third-party and open-source libraries, which might not be component-based. We might have our own legacy libraries to use that are also not component-based. These software libraries, of necessity, must come together at a level of aggregation larger than components, as depicted in Figure 2-3.

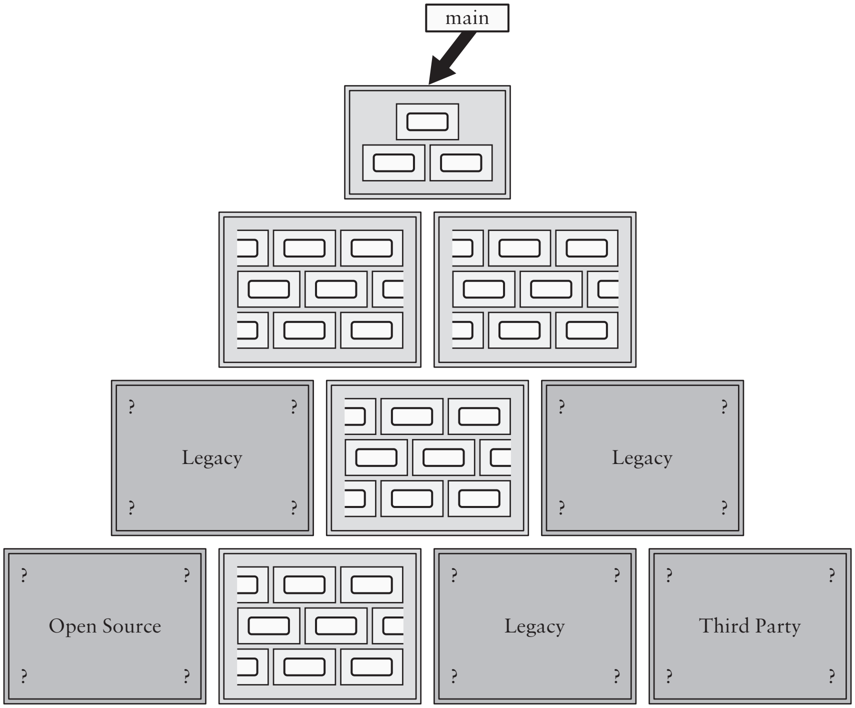


Figure 2-3: Integration with non-component-based (library) software

We generally think of a top-level unit of integration within a large system informally as a "library" whose interface typically consists of a collection of header files in a single directory (e.g., `/usr/include`) and a single library archive (e.g., `libc.a`, `libc.so`) depending on the target platform. We might uniquely refer to this particular *architectural* entity as a whole as "The C Library" although its internal structure (i.e., how logical content is partitioned among its `.o` files) is entirely *organizational* (i.e., not part of its specification or *contract;* see Volume II, section 5.2) and might vary from one vendor platform to another.

Integration with legacy, open-source, and third-party libraries is important and will be addressed. Our purpose in the next few sections, however, is first to identify desirable characteristics of library software and then to provide a prescriptive methodology for packaging our own. After that, we will return to the issues of integrating with non-component-based software (see section 2.12) and then focus on the custom (nonshareable) top-level application code surrounding `main()` (see section 2.13).

2.2 Physical Aggregation

In the preceding chapters, we talked about the atomic unit of physical design, which we call a component, and also the physical hierarchy created by their (acyclic) physical dependencies. Scalability demands hierarchy, and the hierarchy imposed by physical dependency, while of critical importance, is only one architectural aspect of large-scale physical design. Separately, we must also consider how related components can be packaged into larger cohesive physical units. We refer to this other hierarchical dimension of component-based design as *physical aggregation*.

2.2.1 General Definition of Physical Aggregate

DEFINITION: An *aggregate* is a cohesive physical unit of design comprising logical content.

The purpose of aggregation is to bring together logical content (in the form of C++ source code) as a cohesive physical entity that can be treated architecturally as an atomic unit. At one end of the physical-aggregation spectrum lies the component. Each individual component aggregates logical content. Figure 2-4 illustrates schematically a collection of 15 components having 5 separate levels of physical dependency that together might represent a hierarchically reusable subsystem.

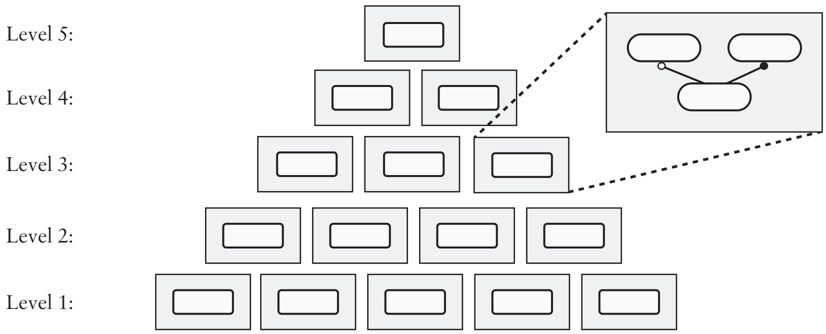


Figure 2-4: Logical content aggregated within 15 individual components

2.2.2 Small End of Physical-Aggregation Spectrum

DEFINITION: A *component* is the innermost level of physical aggregation.

By design, each component embodies a limited amount of code — typically only a few hundred to a thousand lines of source[3] (excluding comments and the component's associated test driver). A single component is therefore too fine-grained (section 0.4) to fully represent most nontrivial architectural subsystems and *patterns*.[4] For example, given a protocol (section 1.7.5) for, say, an (abstract) memory allocator (see Volume II, section 4.10), we might want to provide several distinct components defining various concrete implementations, each tailored to address a different specific behavioral and performance need.[5] Taken as a whole, these components naturally represent a larger cohesive architectural entity, as illustrated in Figure 2-5. To capture these and other cohesive relationships among logically related components — assuming they do not have substantially disparate physical dependencies — we might choose to colocate them within a larger physical unit (see sections 2.8, 2.9, and 3.3). In so doing, we can facilitate both the discovery and management of our library software.

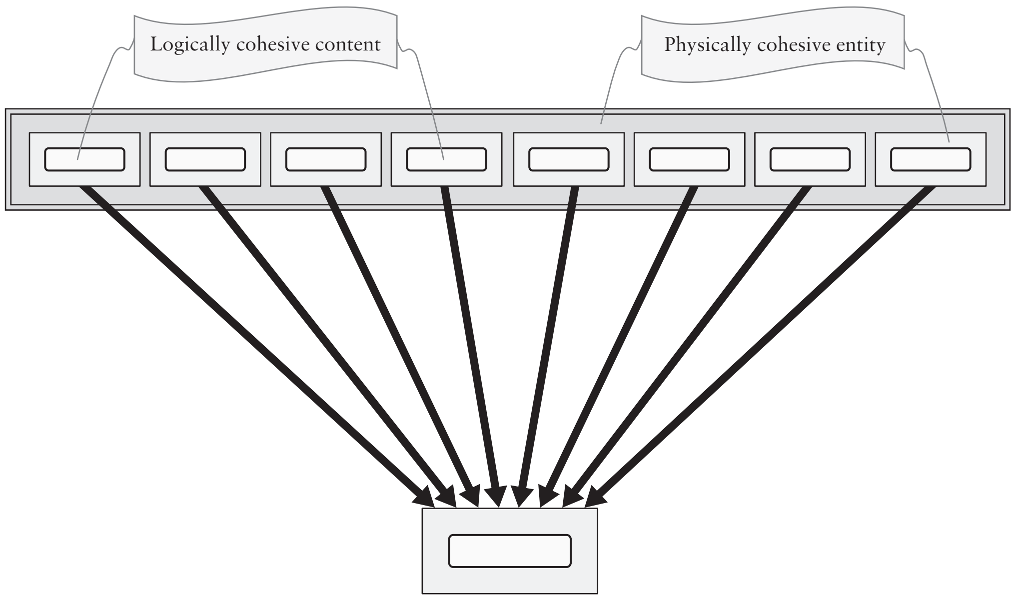


Figure 2-5: Suite of logically similar yet independent components

---

[3] Note that complexity of implementation, coupled with our ability to understand and *test* a given component — more than line count itself — governs its practical maximum "size" (see Volume III, sections 7.3 and 7.5).

[4] See **gamma94**.

[5] E.g., `bdlma::MultipoolAllocator`, `bdlma::SequentialAllocator`, and `bdlma::BufferedSequentialAllocator` (see **bde14**, subdirectory `/groups/bdl/bdlma/`).

2.2.3 Large End of Physical-Aggregation Spectrum

DEFINITION: A *unit of release* (*UOR*) is the outermost level of physical aggregation.

At the other end of the physical-aggregation spectrum is the *unit of release* (UOR), which represents a physically (and usually also logically) cohesive collection of software (source code) that is designed to be deployed and consumed in an all-or-nothing fashion. Each UOR typically comprises multiple separate smaller physical aggregates, bringing together vastly more source code than would occur in any individual component. Even so, we should expect our library software will in time grow to be far too large to belong to any one UOR. Hence, from an enterprise-wide planning perspective, we must be prepared to accommodate the many UORs that are likely to appear at the top level of our inventory of library source code.

2.2.4 Conceptual Atomicity of Aggregates

Guideline

Every physical aggregate should be treated atomically for design purposes.

Even though a UOR may aggregate otherwise physically independent entities, it should nonetheless always be treated, for design purposes, as atomic.[6] Like a component (and every physical aggregate), the granularity with which the contents of a UOR are incorporated into a dependent program will depend on organizational, platform-specific, and deployment details, none of which can be relied upon at design time. Hence, we must assume that any use of a UOR could well result in incorporating all of it — and everything it depends on — into our final executable program. For this reason alone, how we choose to aggregate our software into distinct UORs is vital.

---

[6] The assertion that a library may not be organizationally atomic is true for conventional static (`.a`) libraries (section 1.2.4), but not generally so for shared (`.so`) libraries. Even with static libraries, regulatory requirements (e.g., for trading applications) may force substantial retesting of an application when relinked against a static library whose timestamp has changed, even when the only difference is an additional unused component. In such cases, we may — for the purpose of optimization only — choose to partition our libraries into multiple regions (e.g., multiple `.so` or `.a` libraries) as a post-processing step during deployment (see section 2.15.10). Again, such organizational optimizations in no way affect the architecture, use, or *allowed dependencies* (see section 2.2.14) of the UOR.

2.2.5 Generalized Definition of Dependencies for Aggregates

DEFINITION: An aggregate `y` Depends-On another aggregate `x` if any file in `x` is required in order to compile, link, or thoroughly test `y`.

This definition of physical dependency for aggregates intentionally casts a wide net, so that it can be applied to aggregates that do not necessarily follow our methodology. For aggregates composed entirely of components as defined by the four properties in Chapter 1,[7] the definition of direct dependency of **`y`** on **`x`** reduces to whether any file in **`y`** includes a header from **`x`**.

Observation

The Depends-On relation among aggregates is transitive.

Given the atomic nature with which physical aggregates must be treated for design purposes, if an aggregate **`z`** Depends-On **`y`** (directly or otherwise) and **`y`** in turn Depends-On **`x`**, then we must assume, at least from an architectural perspective, that **`z`** Depends-On **`x`**.

2.2.6 Architectural Significance

DEFINITION: A logical or physical entity is *architecturally significant* if its name (or symbol) is intentionally visible from outside of the UOR in which it is defined.

Architecturally significant entities are those parts of a UOR that are intended to be seen (and potentially used) directly by external clients. These entities together effectively form the *public interface* of the UOR, any changes to which could adversely affect the stability of its clients. The definition of *architectural significance* emphasizes deliberate intent, rather than just the actual physical manifestation, because it is that intent that is necessarily reflected by the architecture.

[7] Component Properties 1–3 (sections 1.6.1–1.6.3) and Component Property 4 (section 1.11.1).

A suboptimal implementation might, for example, inadvertently expose a symbol (at the `.o` level) that was never *intended* for use outside the UOR. If such unintentional visibility were to occur within a UOR consisting entirely of components, it would likely be due to an accidental violation of Component Property 2 (section 1.6.2) and not a deliberate (and misguided) attempt to provide a secret "backdoor" access point. Repairing such defects would not constitute a change in architecture — especially in this case, since any use of such a symbol would itself be a violation of Component Property 4 (section 1.11.1).

2.2.7 Architectural Significance for General UORs

In our component-based methodology, all the software that we write outside the file that implements `main()` is implemented in terms of components. Unfortunately, not all UORs that we might want or need (or be compelled) to use are necessarily component-based (the way we would have designed them). We will start by considering the parts of a general UOR that are architecturally significant irrespective of whether or not they are made up exclusively of components. Later we will discuss the specifics of those that fortunately are.

2.2.8 Parts of a UOR That Are Architecturally Significant

In a nutshell, each externally accessible `.h` file,[8] each nonprivate logical construct declared within those `.h` files, and the UOR itself are all architecturally significant. To make use of logical entities from outside the UOR in which they are defined, their (package-qualified) names (see section 2.4.6) will be needed. In addition, the `.h` files declaring those entities must (or at least should) be included (section 1.11.1) — by name — directly (see section 2.6) for clients to make substantive use of them. Finally, to refer to the particular library comprising the `.o` files corresponding to a UOR (e.g., for linking purposes), it will be necessary to identify it, again, by name.

2.2.9 What Parts of a UOR Are *Not* Architecturally Significant?

While `.h` files are naturally architecturally significant, `.cpp` files and their corresponding `.o` files are not. If we were to change the names of header files or redistribute the logical constructs declared within them, it would adversely affect the stability of its clients; however, such is not the case for `.cpp` or `.o` files. Assuming the UOR is identified in totality by its name, the internal

---

[8] Some methodologies allow for the use of "private" header files (e.g., see Figure 1-30, section 1.4) that are not deployed along with the UOR; our component-based approach (sections 1.6 and 1.11) does not (for good reasons; see section 3.9.7), but does provide for subordinate components (see section 2.7.5).

organization of the library archive that embodies the `.o` files (corresponding to its `.cpp` files) comprised by that UOR will have absolutely no effect on client source code. What's more, changing such *insulated* details (see section 3.11.1) will not require client code even to recompile.

2.2.10 A Component Is "Naturally" Architecturally Significant

For UORs consisting of `.h`/`.cpp` pairs forming components as defined in Chapter 1, both the `.h` and `.cpp` files will each have the component name as a prefix (see section 2.4.6), making components architecturally significant as well. To maximize hierarchical reuse (section 0.4), all components within a UOR and all nonprivate constructs defined within those components are normally architecturally significant. There are, however, valid engineering reasons for occasionally suppressing the architectural significance of a component. Section 2.7 describes how we can — by conventional naming — effectively limit the visibility of (1) nonprivate logical entities outside of the component in which they are defined, and (2) a component as a whole.

2.2.11 Does a Component Really Have to Be a `.h/.cpp` Pair?

What ultimately characterizes a component architecturally is governed entirely by its `.h` file. In Chapter 1, we arrived at the definition of a component as being a `.h`/`.cpp` pair satisfying four essential properties. In virtually all cases, this phrasing serves as *the* definition of a component in C++.[9] For completeness, however, we point out that, though this definition is sufficient and practically useful, it is not strictly necessary. The true essential requirement for components in C++ is that there be *exactly one* `.h` file and one[10] (at least) *or more* (see below) `.cpp` files that together satisfy these four essential properties.

2.2.12 When, If Ever, Is a `.h/.cpp` Pair Not Good Enough?

In exceedingly rare cases,[11] there might be sufficient justification to represent a single component using multiple `.cpp` files. Unlike header files, `.cpp` files in a component, and especially the resulting `.o` files in a statically linked library (`.a`), are not considered *architecturally significant*. For example, a component **`myutil`** defining three logically related, but physically independent functions might reasonably be implemented as having a single header file

---

[9] More generally, for any given language that supports multiple units of translation (e.g., C, C++, Java, Perl, Ada, Pascal, FORTRAN, COBOL), the physical form of a component is standard and independent of its content.

[10] We require that the component header be included in at least one component `.cpp` file so that we can observe, just by compiling the component, that its `.h` file is self-sufficient with respect to compilation (section 1.6.1).

[11] E.g., to further reduce the size of already tiny programs (such as embedded C) or to break hopelessly large (particularly computer-generated) components into separate translation units of a size manageable for the compiler.

`myutil.h` and multiple implementation files — e.g., `myutil.1.cpp`, `myutil.2.cpp`, and `myutil.3.cpp` — each uniquely named, but all sharing the component name as a common prefix. Consequently, a program calling only one of the three functions *might*, under certain deployment strategies (see section 2.15), wind up incorporating only the one `.o` file corresponding to the needed function. Such nuanced considerations are not relevant to typical development and are most usually relegated to the subdomain of embedded systems.

2.2.13 Partitioning a `.cpp` File Is an Organizational-Only Change

It is important to realize that the aggressive physical partitioning discussed above is permissible only because it is *organizational* and not *architectural*. That is, our view and use of the component, its logical design, and its physical dependencies are left unaffected by such architecturally *insignificant* optimizations. Introducing (or removing) such optimizations has no effect on the client-facing interface (including any need for recompilation) or logical behavior, only on program size. By contrast, introducing multiple `.h` files for a single component would represent an architectural change manifestly affecting usage; hence, a component — in all cases — *must* have exactly one header file, whose root name identifies the component *uniquely* (see section 2.2.23).

2.2.14 Entity Manifest and Allowed Dependencies

DEFINITION: A *manifest* is a specification of the collection of physical entities — typically expressed in external metadata (see section 2.16) — intended to be part of the physical aggregate to which it pertains.

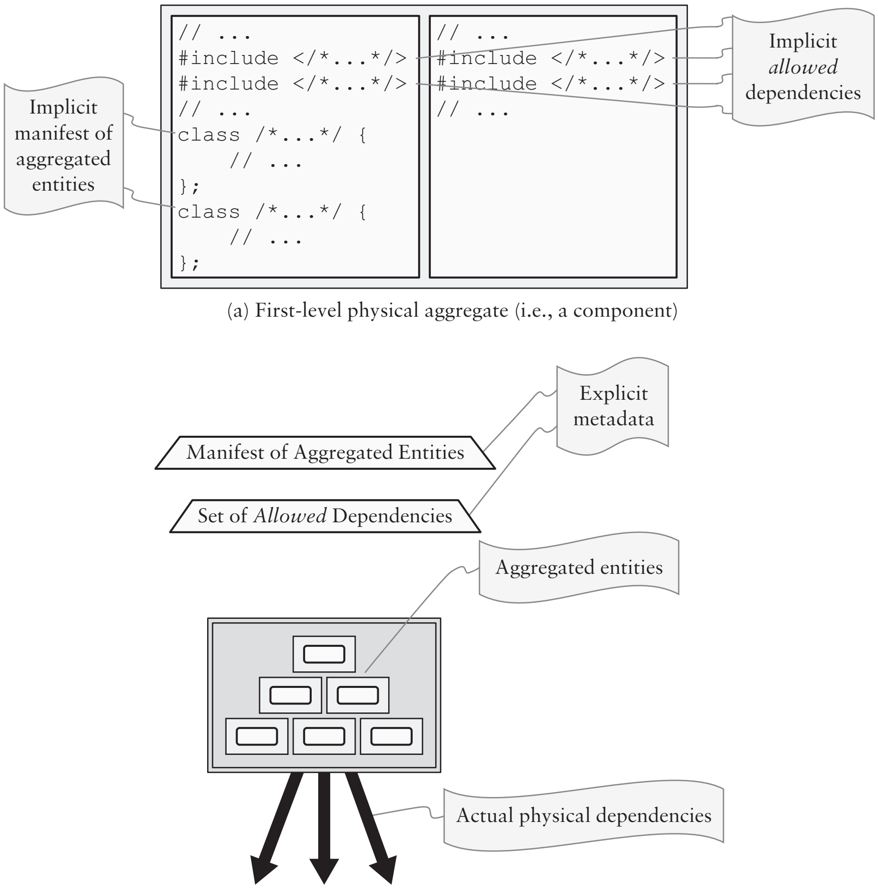
DEFINITION: An *allowed dependency* is a physical dependency — typically expressed in external metadata (see section 2.16) — that is permitted to exist in the physical hierarchy to which it pertains.

Observation

The definition of every physical aggregate must comprise the specification of (1) the entities it aggregates, and (2) the external entities that it is *allowed* to depend on *directly*.

To be practically useful, every aggregate (from a component to a UOR) must, at a minimum, somehow allow us to specify contractually the entities it aggregates, as well as the other physical entities upon which those contained entities are *allowed* (i.e., explicitly permitted) to depend directly. Much of our design methodology is anchored in understanding the physical dependencies among the discrete *logically and physically cohesive* (see section 2.3) entities within our software. Given a dependency graph, without knowing the specific (outwardly visible) entities at its nodes or its (permissible) edges, there is simply no good way to reason about it.

For any given component, as illustrated in Figure 2-6a, the manifest of aggregated entities is implied by the accessible logical entities declared within its header file. The *allowed* direct dependencies are implied by the combined `#include` directives embedded within the `.h` and `.cpp` files of that component (section 1.11). For the second and successive levels of physical aggregation, the manifest of member aggregates and list of *allowed* dependencies is an essential part of the architectural specification and must somehow be stated explicitly (Figure 2-6b).



(a) First-level physical aggregate (i.e., a component)

(b) Second-level physical aggregate

Figure 2-6: Specifying members and *allowed* dependencies for aggregates

Unfortunately, the C++ language itself does not support any notion of architecture beyond a single translation unit.[12] Hence, much of the aggregative structure we discuss in this chapter will have to be implemented alongside the language using metadata (see section 2.16). This metadata will be kept locally as an integral part of each aggregate to help guide the tools we use to develop, build, and deploy our software.[13] An abstract subsystem consisting of four second-level aggregates forming three separate (aggregate) dependency levels is illustrated schematically in Figure 2-7.

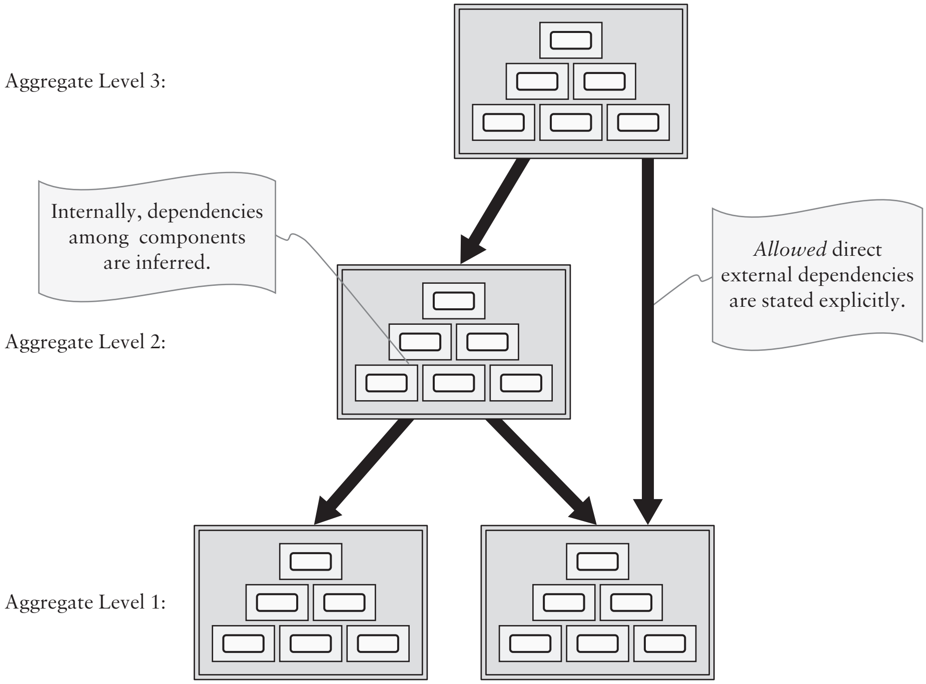


Figure 2-7: Schematic subsystem built from second-level physical aggregates

[12] As of this writing, work was progressing in the C++ Standards Committee to identify requirements for a new packaging construct called a `module` (see **lakos17a** and **lakos18**), and a preliminary version of this long-anticipated *modules* feature was voted into the draft of the C++20 Standard at the committee meeting in Kona, HI, on February, 23, 2019.

[13] A detailed overview of this architectural metadata along with its practical application and how build and other tools might consume it is provided for reference in section 2.16.

2.2.15 Need for Expressing Envelope of Allowed Dependencies

Expressing the envelope of *allowed* dependencies for aggregations of components explicitly might, at first, seem redundant and therefore unnecessary. As noted in section 1.11, there are numerous dependency-analysis tools available that can be used to extract actual dependencies from the aggregated components and produce the envelope of those dependencies across physical aggregates automatically, but to do so misses the point: The purpose of stating *allowed* dependencies is to be anticipatory, not reactive. Characterizing a set of proposed aggregations and then supplying an envelope of *allowed* dependencies among those aggregations enables us to express our physical design (intent) *before* any code is written. As new functionality is added, unexpected physical dependencies can be detected and flagged as implementation errors. Without specifying *allowed* dependencies *a priori*, there is no physical design to implement, let alone verify. Hence, explicitly specifying — and verifying — *allowed* dependencies is necessary at every level of physical aggregation.

2.2.16 Need for Balance in Physical Hierarchy

Observation

To maximize human cognition, peer entities within a physical aggregate should be of comparable physical complexity (e.g., have the same level of physical aggregation).

Between a component and a UOR, we might imagine that there could (in theory) be any number of intermediate levels of physical aggregation, each of which might or might not have architectural significance. Some physical aggregation hierarchies are better than others. In particular, an unbalanced hierarchy, such as the one illustrated schematically in Figure 2-8, is suboptimal.

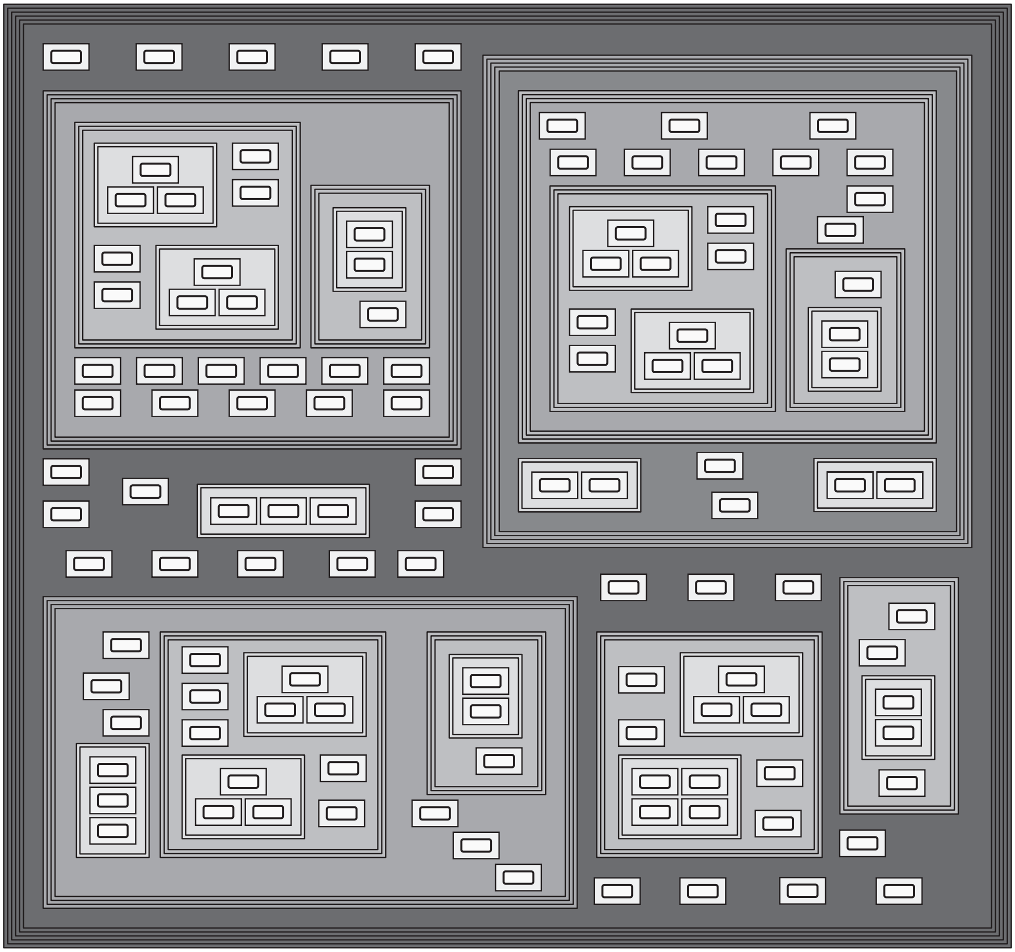
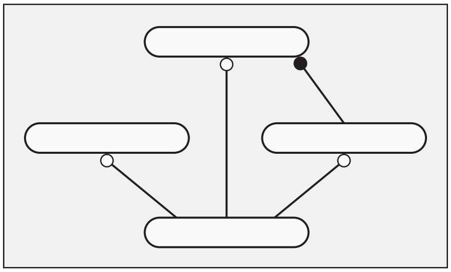


Figure 2-8: UOR having unbalanced levels of physical aggregation (BAD IDEA)

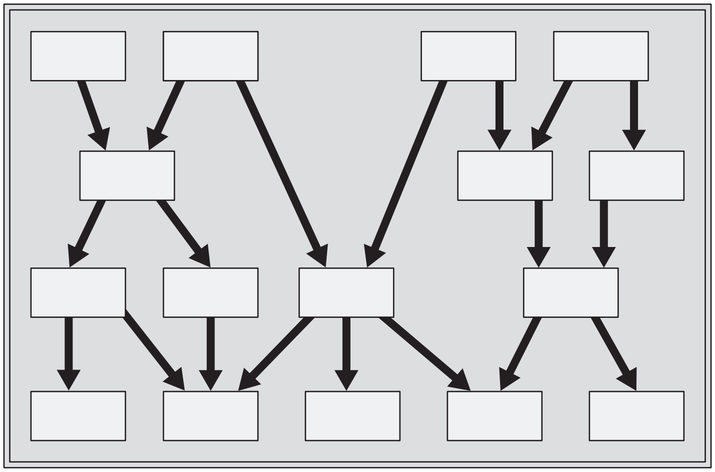
2.2.17 Not Just Hierarchy, but Also Balance

Effective regular decomposition of large systems requires not only hierarchy, but also balance. We choose to model our software development accordingly. Although not strictly necessary, we want each aggregate to comprise entities having similar physical complexity. In particular, we deliberately avoid placing components alongside larger aggregates within a UOR. We find that entities having comparable complexity at each aggregation depth improves comprehension and facilitates reuse.

At each increasing level of physical aggregation, we strive to bring together a significant, but not overwhelming amount of information and engineering at a uniform level of abstraction such that it can be understood and used effectively. As a rule, we would like the relevant schematic detail to correspond to what might reasonably fit on a single 8 1/2 × 11 inch piece of paper[14] as suggested by the complexity of each of the individual diagrams in Figure 2-9. By achieving this balance — much like the chapters and sections within this book — we provide fairly uniformly chunked content, which makes it more convenient to analyze and discuss.

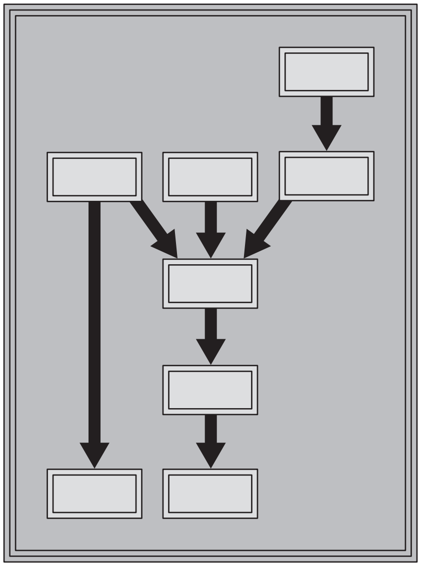


(a) Aggregation level I: component containing related logical content



(b) Aggregation level II: package of related components

[14] Being an American, I have chosen the most common loose-leaf paper size in the United States, as opposed to ones conforming to ISO 216 used by other countries where A4 is the most common (and similar) size (see http://www.papersizes.org/).



(c) Aggregation level III: group of related packages

Figure 2-9: Balancing complexity at each level of physical aggregation

2.2.18 Having More Than Three Levels of Physical Aggregation Is Too Many

Observation

More than three levels of appropriately balanced physical aggregation are virtually always unnecessary and can be problematic.

While components (being deliberately fine grained) are too small to be practical to release or deploy individually, having more than three appropriately balanced levels of physical aggregation (as illustrated schematically in Figure 2-10) is not especially useful and can be impractical due to the sheer magnitude of the code involved. There are limits as to what we can reasonably fit into a single physical library and what typical development and build tools can accommodate. There are also design and deployment issues that would tend to discourage physically aggregating such massive architectural entities.

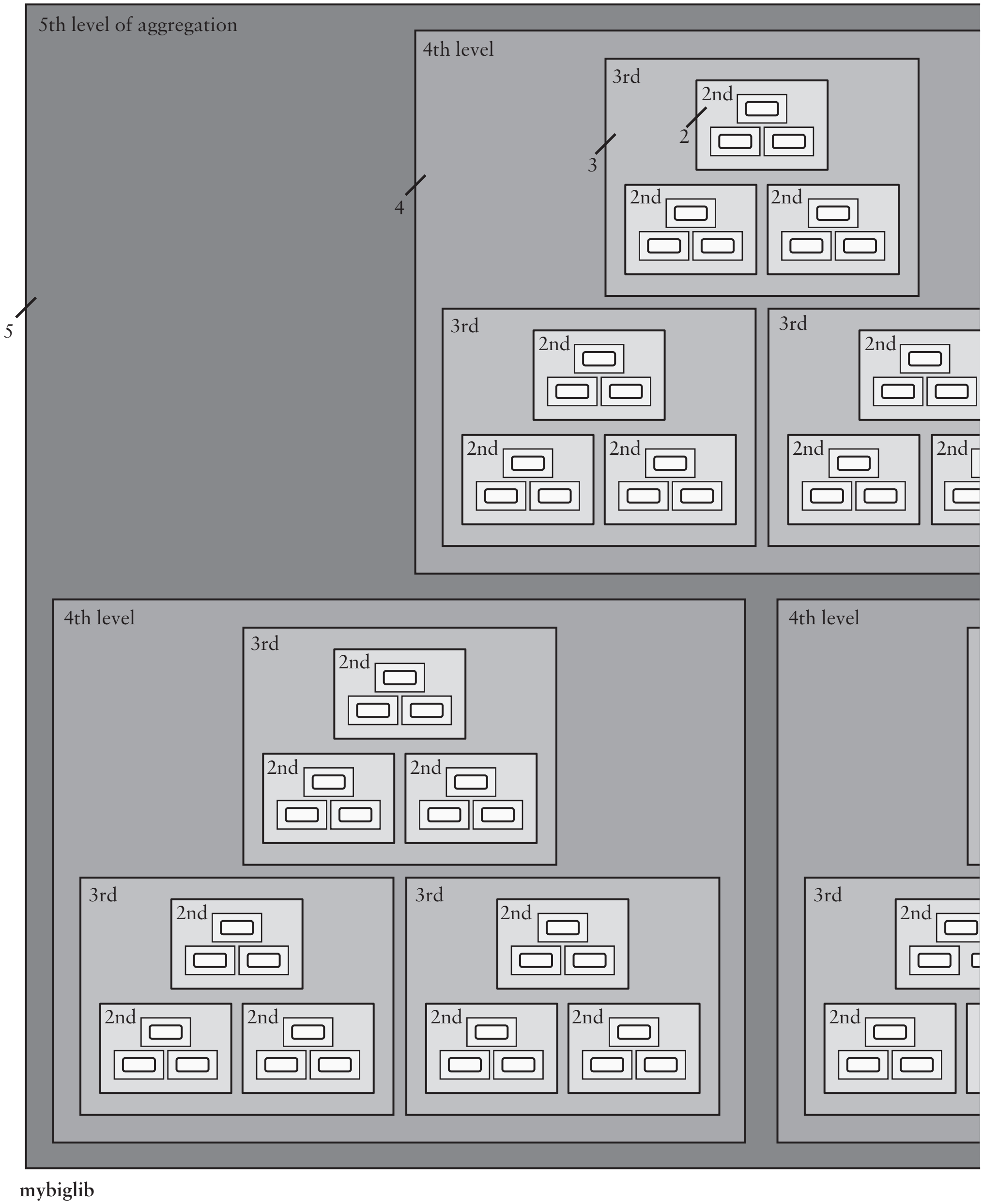


Figure 2-10: More than three levels of physical aggregation (BAD IDEA)

2.2.19 Three Levels Are Enough Even for Larger Systems

In our experience, we find that three appropriately balanced, architecturally significant levels of physical aggregation have been sufficient to represent very large libraries. When there are three architecturally significant levels, we will consistently refer to each entity at the second level of architecturally significant aggregates within the UOR as a *package*[15] (see section 2.8) and the UOR itself as a *package group* (see section 2.9).

For example, using even the modest size estimates for a component, package, and package group illustrated in Figure 2-11, each UOR would, on average, support a couple of hundred thousand lines of noncommentary source code — excluding, of course, the corresponding component-level test drivers (see Volume III, section 7.5). Thus, an enterprise-wide body of library software consisting of 10 million lines of source code could fit comfortably within fifty such UORs, with yet larger code bases requiring only proportionately more.

$$500\,\frac{\text{source lines}}{\text{component}} \times 20\,\frac{\text{components}}{\text{package}} \times 20\,\frac{\text{packages}}{\text{package group}} = 200{,}000\,\frac{\text{source lines}}{\text{UOR}}$$

Figure 2-11: Modest size estimates of components, packages, and package groups.

2.2.20 UORs Always Have Two or Three Levels of Physical Aggregation

Hence, in our methodology, the number of appropriately balanced, architecturally significant levels of physical aggregation within our library software will always be at least two (i.e., the individual components and the UOR that comprises them), but never more than three.

There might, in rare cases, be valid reasons — e.g., to accommodate a large, monolithic, externally designed interface[16] — to introduce, purely for organizational purposes, an additional, intervening level of physical aggregation. Any such organization-based partitioning of the implementation of an architecturally significant aggregate — just like with that of a component — should, of course, never be architecturally significant (see section 2.11).

---

[15] Note that a UOR can also be an isolated package, but there should be a compelling engineering reason for preferring to do so over a package group, especially for (hierarchically reusable) library software.

[16] The C++ Standard Library residing entirely in the `std` namespace, is itself an example of such a monolithic specification.

2.2.21 Three Balanced Levels of Aggregation Are Sufficient. Trust Me!

The "artificial" constraints on physical aggregation suggested here do not in any way stop individual developers from being creative; rather, this regularly structured physical aggregation model helps to focus creativity where it will be most effective — the functionality, not the packaging — thereby making our software developers as a whole more successful. It will turn out that having a regular, balanced, and fairly shallow architectural structure also lends itself to an economical notation for identifying every architecturally significant logical and physical entity within our proprietary library software (see section 2.4).

2.2.22 There Should Be Nothing Architecturally Significant Larger Than a UOR

We deliberately avoid creating anything architecturally significant that is larger than a single (physical) UOR.[17] Treating such expansive *logical* units atomically, as illustrated in Figure 2-12a, would increase our envelope of allowed dependencies without providing any concrete encapsulation of logical functionality within a cohesive physical entity (see section 2.3). Instead, we choose to model such coarse architectural policy more articulately as individual *allowed* physical dependencies among UORs (Figure 2-12b). The more that we can encapsulate each logical subsystem within a single (architecturally significant) physical aggregate, the more we will be able to infer useful physical dependencies (section 1.9) from logical relationships across those entities.

---

[17] Having a single, enterprise-wide namespace in which to guard the names within *all* of the components we collectively write is (1) independent of any aspect of specific designs, and (2) a good idea (see section 2.4.6).

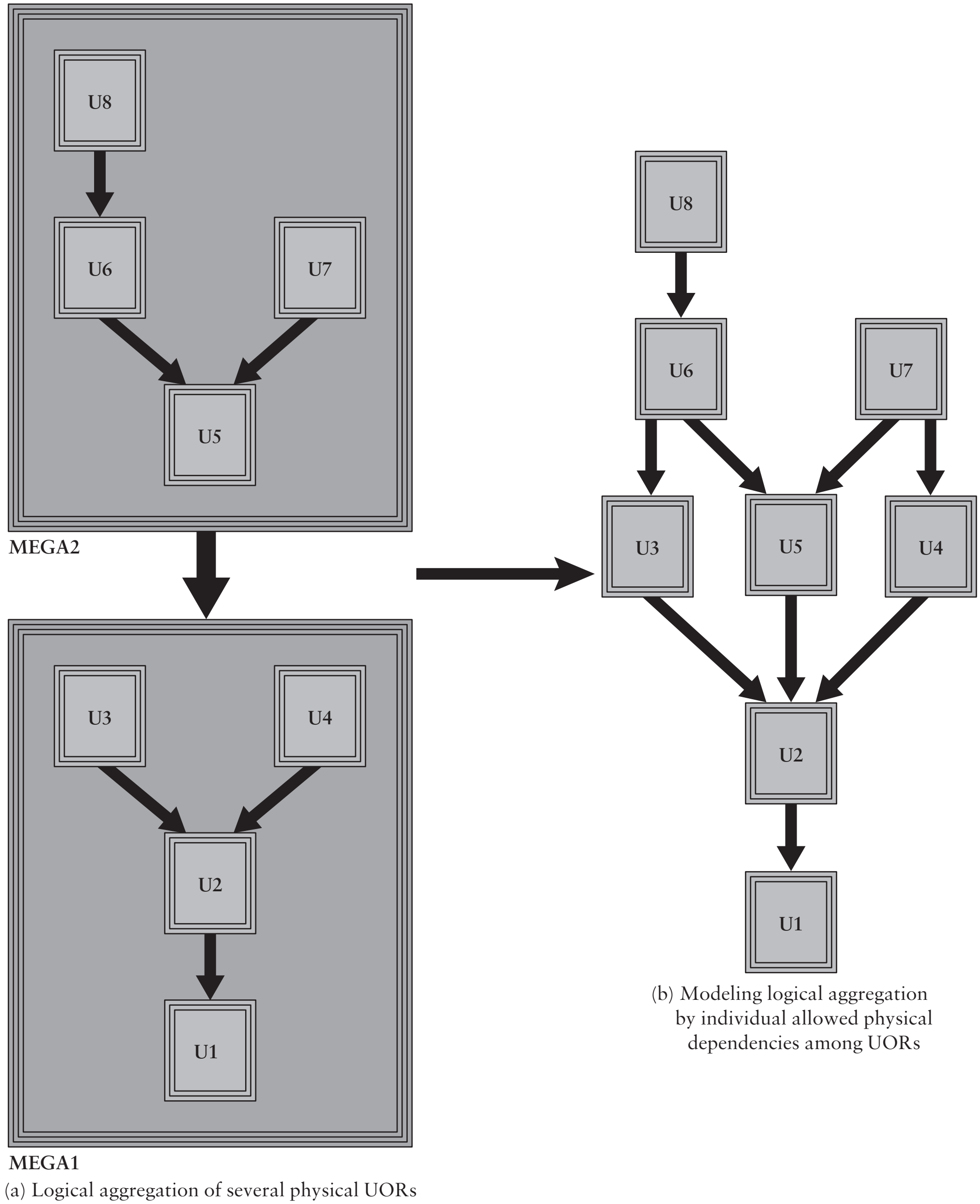


Figure 2-12: Supplanting logical aggregation with allowed physical dependency

2.2.23 Architecturally Significant Names Must Be Unique

Design Rule

The name of every architecturally significant entity must be unique throughout the enterprise.

The C++ language requires that the name of every logical entity visible outside of the translation unit in which it is defined must be unique within a program (section 1.3.1). We need more. We require that the names of all externally accessible logical entities within our library identify each entity uniquely because, with reuse, a combination of those logical entities might one day wind up within the same program (see section 3.9.4). For the same reason, the names of all UORs (package groups and packages) and components — each also being visible to external clients — must be globally unique as well.

Even without our cohesive naming strategy (see section 2.4), there remain compelling advantages (e.g., see sections 2.4.6 and 2.15.2) to ensuring that component filenames are themselves guaranteed to be globally unique throughout the enterprise — irrespective of directory structure.[18]

> The benefit of unique filenames is uniqueness. When one sees a filename (such as `xyza_context.h`) anywhere in the system — be it in a log message, an assertion, an email, or a tab in a text editor – one knows, uniquely, the component to which it refers. Unique filenames also make the rendering of include directives in source code orthogonal to the physical placement of headers on a filesystem. A lack of unique filenames does not break any one thing, but makes a large collection of tasks more difficult because the filename itself is no longer a unique identifier. In a large-scale organization with hundreds of thousands of components (among which there will inevitably be many having the *base name* "context"), maintaining the filename as a unique identifier has been, and will continue to be, a very valuable property indeed!
>
> — Mike Verschell

[18] On April 1, 2019, Mike Verschell became the manager of Bloomberg's BDE team, replacing its founder (John Lakos) after nearly eighteen rewarding years of applying the methodology described in this book to developing real-world large-scale C++ software. Mike provided the quoted synthesis of his position on unique filenames via personal email.

2.2.24 No Cyclic Physical Dependencies!

Design Imperative

Allowed (explicitly stated) dependencies among physical aggregates must be acyclic.

Cyclic physical dependencies[19] among any physical entities — irrespective of the level of physical aggregation — do not scale and are always undesirable. Such cyclically interdependent architectures are not only harder to build, they are also much, much harder to comprehend, test, and maintain than their acyclic counterparts. In fact, to help improve human cognition, we almost always structure our source code to avoid forward references to logical entities even within the same component. Whenever the physical specification of a design would allow cyclic dependencies among architecturally significant physical aggregates, we assert that the design is unacceptably flawed. Even if, for some unusual (organizational) reason, we were to choose to partition an outwardly visible aggregate into subaggregates that were *not* architecturally significant (e.g., see section 2.11), we would nonetheless insist that the allowed dependencies among those subaggregates be acyclic as well (see also Figure 2-89, section 2.15.10).

2.2.25 Section Summary

In summary, a physical aggregate is a physically cohesive unit of logical content and a necessary abstraction in any development process. The organizational details of a physical aggregate will likely vary from one platform, compiler/linker technology, and deployment strategy to the next; hence, each physical aggregate is treated, at least architecturally, as atomic. Our logical designs must also, therefore, always be governed by the envelope of architecturally *allowed* (rather than actual) physical dependencies specified for the aggregate. Balancing complexity at each successive level of aggregation facilitates human cognition and potential reuse. The use of three balanced levels of architecturally significant physical aggregation has been demonstrated to be sufficient (and in fact optimal) to describe even the largest of systems. We do, however, want to avoid architecturally significant logical entities (other than an enterprise-wide namespace) that span UORs.

[19] A collection of interdependent (connected) entities is cyclically dependent if the transitive closure of the binary relation matrix representing direct dependencies between any two entities is not antisymmetric.

2.3 Logical/Physical Coherence

When developing large-scale software, it is essential that our logical and physical designs coincide in several fairly specific ways at every level of packaging. Perhaps the most fundamental property of well-packaged software is that all logical constructs advertised within the collective interface of a physical module or aggregate — e.g., component, package, UOR (section 2.2) — are implemented directly within that module. Software that does not have this property generally cannot be described in terms of a graph where the nodes represent cohesive *logical* content and the directed edges represent (acyclic) dependencies on other *physical* modules. We refer to such undesirable software as *logically and physically incoherent*.

For example, Component Property 3 (section 1.6.3) states that if a logical construct having external bindage is declared in a component's header, then that component is the only one permitted to define that construct. Recall from section 1.9 that, knowing the logical relationships among classes contained within separate components having Component Property 3, we can reliably infer physical dependencies among those components. Arbitrary `.h`/`.cpp` pairs that do not fully encapsulate the definitions of their logical constructs unnecessarily make reasoning about the design (and organizational) dependencies substantially more complicated (e.g., the misplaced definition of the output operator for the `Date` class in Figure 1-46, section 1.6.3). We therefore require that whatever logical constructs a component advertises as its own are defined entirely within that component, and never elsewhere.

Guideline

Architecturally cohesive logical entities should be tightly encapsulated within physical ones.

The same benefits of logical/physical coherence that we derive from individual components apply also to library software at higher levels of aggregation. Imagine, for example, that we have two fairly large logical subsystems that we call **`buyside`** and **`sellside`**. Each subsystem is composed of several classes. For this discussion, let us assume that each of the classes is defined in its own separate component, and that the dependency graph of the unbundled components is acyclic. Figure 2-13 shows what often happens when subsystems conceived from only a logical perspective materialize. Although the logical and physical aspects of these systems coincide, the cyclic physical nature of the aggregate design does not scale, and is therefore unacceptable (section 2.2.24).

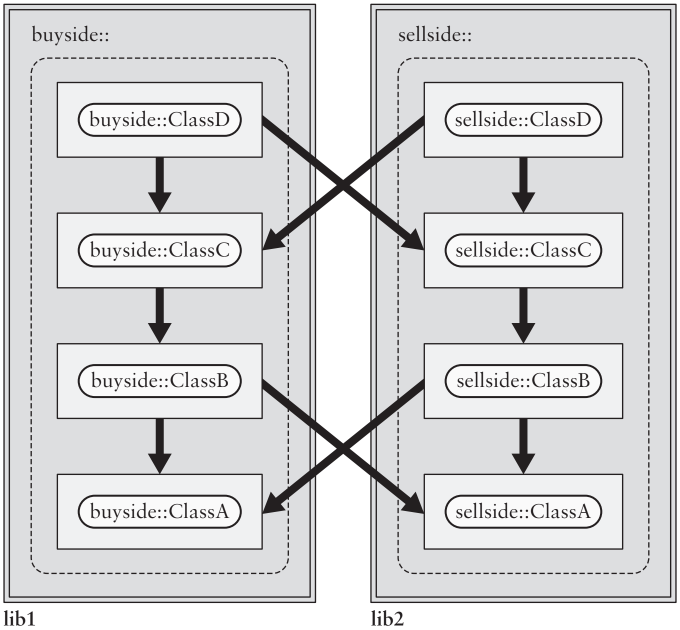


Figure 2-13: Cyclic physical dependencies (BAD IDEA)

Avoiding cyclic physical dependencies across aggregate boundaries is not only for the benefit of build tools, it also facilitates human cognition and reasoning. If all that were needed was to have two libraries where the envelope of component dependencies across aggregates was acyclic, then it would suffice to mechanically repartition these components as shown in Figure 2-14. But for software packaging to facilitate human cognition, in addition to being physically acyclic, the logical and physical aspects of a design must remain *coherent*.

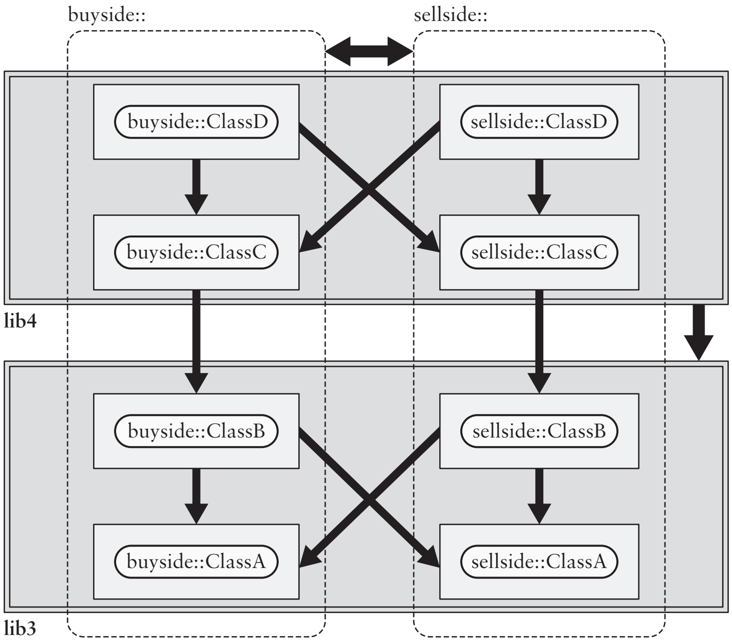


Figure 2-14: Logical/physical *incoherence* (BAD IDEA)

Although the cyclic physical dependencies between the two libraries have been eliminated, the logical and physical designs have diverged. Now, neither logical subsystem is encapsulated by either physical library. As a result, our ability to infer aggregate physical dependencies from abstract logical usage — i.e., at the subsystem level — is lost. That is, if a client abstractly uses *either* the **`buyside`** *or* **`sellside`** logical subsystems, we must either know the details of that usage or otherwise assume an implied physical dependency on *both* libraries. Just as with cyclic physical dependencies, our ability to *reason* about logically and physically incoherent designs does not scale; hence, such designs are to be avoided.

Uniting the logical and physical properties of software is what makes the efficient development of large-scale systems possible. Achieving an effective modularization of logical subsystems is not always easy and might require significant adjustment to the logical design of our subsystems (see Chapter 3). As Figure 2-15 suggests, the reworked design might even yield a somewhat different logical model. Achieving designs having both logical/physical coherence and acyclic physical dependencies early in the development cycle requires forethought but is far easier than trying to tweak a design after coding is underway. Once released to clients, however, the already arduous task of re-architecting a subsystem will invariably become qualitatively more intractable, often insurmountably so.

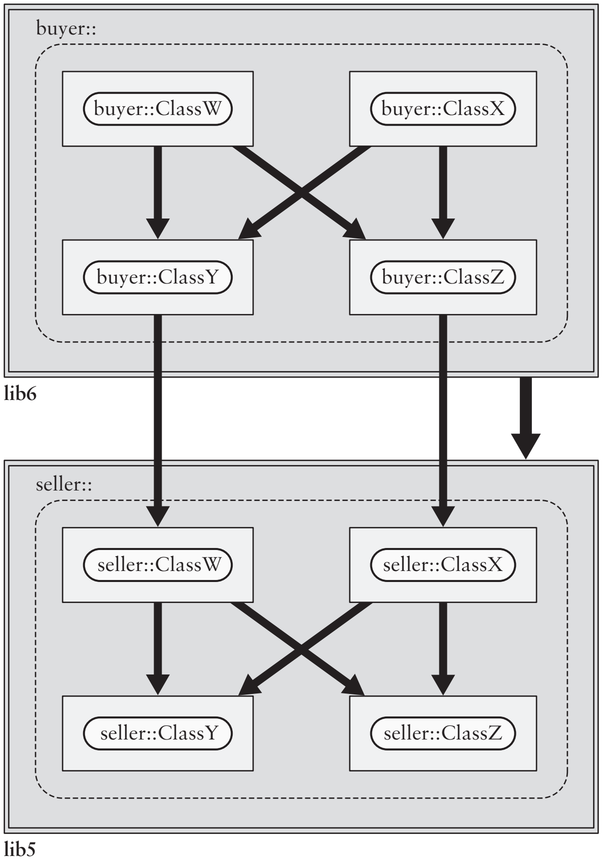


Figure 2-15: Acyclic logical/physical *coherence* (GOOD IDEA)

Achieving logical and physical coherence along with acyclic physical dependencies across our entire code base is absolutely essential. In addition to ensuring these important properties, however, we will need a strategy that guarantees not just that the name of each architecturally significant logical and physical entity is unique throughout the enterprise, but that it can also be identified (and its definition located) just from its point of use, without having to resort to tools (e.g., an IDE). The following section addresses how we realize these additional goals in practice.

2.4 Logical and Physical Name Cohesion

The ability to identify the physical location of the definition of essentially every logical construct — directly from its point of use — is an important aspect of design that distinguishes our methodology from others used in the software industry. The practical advantages of this aspect of design, however, are many and are explored in this section.

2.4.1 History of Addressing Namespace Pollution

Global namespace pollution — specifically, local constructs usurping short common names — is an age-old problem. All of us have learned that naming a class `Link` or a function `max` at file scope — even in a `.cpp` file — is just asking for trouble. Left unmanaged, the probability of name conflicts increases combinatorially with program size. Developers have traditionally responded to this problem with ad hoc conventions for naming logical constructs based on what are *hopefully* unique prefixes (e.g., `ls_Link`, `myMax`, `size_t`). When the use of a logical construct is confined to a single `.cpp` file, we can always make individual functions `static` and nest local classes within the unnamed namespace. The problem of name collisions, however, extends to header files as well.

2.4.2 Unique Naming Is Required; Cohesive Naming Is Good for Humans

Recall from section 2.2.6 that a logical or physical entity is *architecturally significant* if its name (or symbol) is intentionally visible from outside of the UOR that defines it. To refer to each architecturally significant entity unambiguously, we require the name of each such entity to be globally unique. How we achieve this uniqueness is, to some extent, an implementation detail — at least from the compiler's perspective. When it comes to human beings, however, cohesive naming, as we will elucidate in this section, has proven to provide powerful cognitive reinforcement.

Suppose we want to implement an architecturally significant type, say one that represents a *price* — e.g., for a financial instrument. How should we ensure that the name of this type is globally unique? In theory, there are many ways to achieve unique naming. We could, for example, maintain a central registry of logical names. The first developer to choose `Price` gets it! The next developer implementing a similar concept (there are many ways to characterize a price) would be forced to choose something else (e.g., `MyPrice`, `Price23`). The same approach could just as easily be used to reserve unique filenames.

2.4.3 Absurd Extreme of Neither Cohesive nor Mnemonic Naming

Taking this approach to the extreme, we could even have the registry generate unique type names based on a global counter — e.g., `T125061`, `T125062`, `T125063`, and so on. We could do similarly for component names (e.g., **`c05684`**, **`c05685`**, **`c05686`**) and even for units of release (e.g., **`u1401`**, **`u1135`**, **`u1564`**), as illustrated in Figure 2-16. It all works just fine as far as the compiler and linker are concerned. Moreover, physically moving a component from one aggregate to another would have no nominal implications. Human cognition, however, is not served by this approach.

```
// c27341.h              // component defining our "date" class

#include <c11317.h>      // Declares T161459 implementing day-of-week.

// ...

class T121056;           // Local Declaration of In-Stream Facility
class T121059;           // Local Declaration of Out-Stream Facility

class T121547 {          // definition of our "date" class

    static bool isYearMonthDayValid(int year, int month, int day);

    // ...

    T121547();
    T121547(int year, int month, int day);
    T121547(const T121547& original);
    ~T121547();

    // ...

    T121547& operator=(const T121547& rhs);

    // ...

    void setYearMonthDay(int year, int month,int day);
    int setYearMonthDayIfValid(int year, int month, int day);

    // ...

    int year() const;
    int month() const;
    int day() const;
    T161459::Enum dayOfWeek() const;

};

// ...

T121056& operator>>(T121056&  inStream,       T121547& date);
T121059& operator<<(T121059& outStream, const T121547& date);
```

Figure 2-16: Absurdly opaque, noncohesive generated unique names (BAD IDEA)

Maintaining a central database to reserve individual class or component names is not practical and clearly not the best answer. Instead, we will exploit hierarchy to allocate multiple levels of namespaces at once. This hierarchy, however, is neither ad hoc nor arbitrary; with the exception of an overarching enterprise-wide namespace (see below), each namespace that we employ in our methodology will correspond to a coherent, *architecturally significant*, logically and physically cohesive aggregate.

2.4.4 Things to Make Cohesive

For every architecturally significant logical entity there are at least three related architectural names:

1. The name (or symbol) of the logical entity itself
2. The name of the component (or header) that declares the logical entity
3. The name of the UOR that implements the logical entity

Ensuring that these names are deliberately cohesive will have significant implications with respect to development and maintenance. Hence, how and at what physical levels we achieve nominal cohesion is a distinctive and very important design consideration within our methodology.

2.4.5 Past/Current Definition of Package

DEFINITION: A *package* is the smallest architecturally significant physical aggregate larger than a component.

COROLLARY: The name of each package must be unique throughout the enterprise.

A package (see section 2.8) is an *architecturally significant* — i.e., globally visible — unit of logical and physical design that serves to aggregate components, subject to explicitly stated, *allowed dependency* criteria (section 2.2.14). A package is also a means for making related components physically and, as we are about to see, nominally cohesive. In these ways, packages enable designers to capture and reflect, in source code, important architectural information not easily expressed in terms of components alone.

Historically,[20] a package was defined as a collection of components organized as a (logically and) physically cohesive unit (see section 2.8.1). Although every package we write ourselves

[20] **lakos96**, section 7.1, pp. 474–483

will necessarily be implemented exclusively in terms of components, other kinds of well-reasoned architecturally significant physical entities comprising multiple header files, yet not aggregating components, are certainly possible.[21]

With the definition as worded above, the word *package* can serve as a unifying term to describe any architecturally significant body of code that is larger than a component, but without necessarily being component-based. We will, however, consistently characterize packages that are not composed entirely of components adhering to our design rules — especially those pertaining to our cohesive naming conventions delineated throughout the remainder of this section (section 2.4) — as *irregular* (see section 2.12).

Suppose now that we have a logical subsystem called the *Bond Trading System* (referred to in code as `bts` for short). Suppose further that this logical subsystem consists of a number of classes (including a price class) that have been implemented in terms of components, which, in turn, have been aggregated into a package to be deployed atomically as an independent library (e.g., `libbts.a`). How should we distinguish the `bts` *bond* price class from other price classes, and what should be the name of the component in which that price class is defined?

2.4.6 The Point of Use Should Be Sufficient to Identify Location

Guideline

The *use* of each logical entity declared at package-namespace scope should alone be sufficient to indicate the component, package, and UOR in which that entity is defined.

Whenever we see a logical construct used in code, we want to know immediately to which component, package, and UOR it belongs. Without an explicit policy to do otherwise, the name

[21] Robert Martin is the only other popular author we know of to describe in terms of C++ (previous to **lakos96** or otherwise) an even remotely similar concept. In his adaptation of Booch's *Class Categories*, which originally were themselves just logical entities (**booch94**, section 5.1, "Essentials: Class Categories," pp. 581–584), Martin's category unites a cluster of classes related by both logical and physical properties. Based on personal (telephone) correspondence (c. 2005), his augmented categories were intended to be significantly larger than a component, but somewhat smaller than a typical package (see Figure 2-11, section 2.2.19), virtually always sporting exactly one class per header (see section 3.1.1); see **martin95**, "High-Level Closure Using Categories," pp. 226–231.

of a class, the header file declaring that class, and the UOR implementing that class might all have unrelated names, as illustrated Figure 2-17. Clients reading `BondPrice` will not be able to predict, from usage alone, which header file defines it, nor which library implements it; hence, global search tools would be required during all subsequent maintenance of client code.

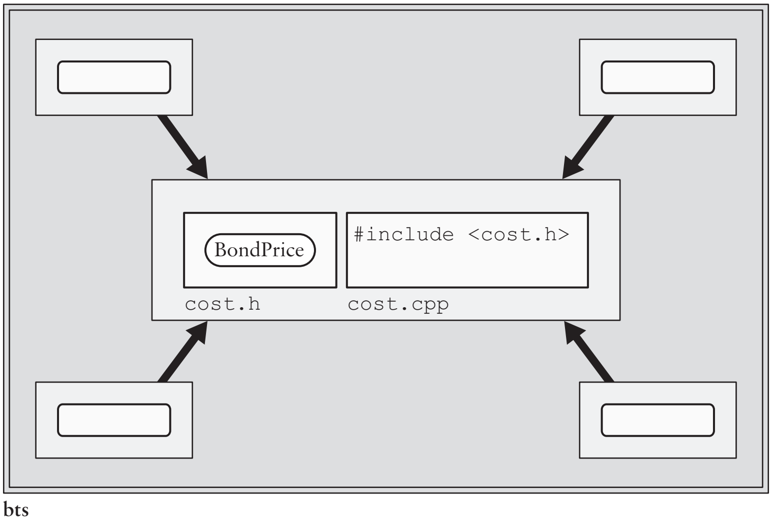


Figure 2-17: Noncohesive logical and physical naming (BAD IDEA)

By the same token, other components packaged together to implement this logical subsystem might well have names that are unrelated to each other, obscuring the cohesive physical modularity of this subsystem. Although not strictly necessary, experience shows that human cognition is facilitated by explicit "visual" associations within the source code. This nominal cohesion, in turn, reinforces the more critical requirement of logical/physical coherence (section 2.3). Hence, logical and physical name cohesion across related architecturally significant entities is an explicit design goal of our packaging methodology.

Design Rule

Component files (`.h`/`.cpp`) must have the same root name as that of the component itself (i.e., they differ only in suffix).

By their nature, components implemented as `.h`/`.cpp` pairs naturally already exhibit some degree of physical name cohesion. Note that as recently as the writing of my first book (1996), however, such was not the case. Due to unreasonable restrictions on the length of names that could be accommodated to distinguish `.o` files contained in library archive (`.a`) files of the day, `.o` files often had to be shortened; hence, an external cross-reference needed to be maintained in order to reestablish the cohesive nature of components.[22]

COROLLARY: Every library component filename must be unique throughout the enterprise.

Recall from section 2.2.23 that every globally visible physical entity must itself be uniquely named. Since library component headers are at least potentially (see section 3.9.7) clearly visible from outside their respective units of release, and their corresponding `.cpp` file(s) derive from the same root name and yet are distinct among themselves, they too must be globally unique. Note that, unlike library components, the names of components residing in application packages (see section 2.13) do not have to be distinct from those in other application packages so long as their logical and physical names do not conflict with those in our library as, in our methodology, no two such application packages would ever be present in the same program.

Design Rule

Every component must reside within a package.

Components, which are intended to address a highly focused purpose and are tailored to bolster hierarchical reuse (section 0.4), are invariably too fine grained to be practical to be released individually (section 2.2.20). Hence, in our methodology, each component is necessarily nested within a higher-level, architecturally significant aggregate, which (by definition) is a *package*. Although the benefits of physical uniformity — enhanced understandability and facilitation of automation tools — as outlined in section 0.7 alone are compelling, mindless adherence to this

[22] **lakos96**, Appendix C, pp. 779–813 and, in particular, Appendix C.1, pp. 180–193

rule, however, will fall far short of the potential benefit it seeks to motivate. The intent here is not just to provide a uniform and balanced physical representation of software, but also to craft a hierarchical repository where the contained elements, from a logical as well as a physical perspective, are cohesive and synergistic (see section 2.8.3). Moreover, we want to ensure that each library component we write has a natural and obvious place in the physical hierarchy of our firm-wide repository (see sections 3.1.4 and 3.12).

Design Rule

The (all-lowercase) name of each component must begin with the (all-lowercase) name of the package to which it belongs, followed by an underscore (_).

A first step toward ensuring overt visible cohesion between architecturally significant names is making sure that the component name reflects the name of the package in which it resides, as shown in Figure 2-18. Just by looking at the name of the **`bts_cost`** component, we know that there exist two component files named `bts_cost.h` and `bts_cost.cpp`, which reside in the **`bts`** package.[23,24]

[23] In our methodology, packages (see section 2.8) are either aggregated into a group (see section 2.9) or else released as standalone packages, with these two categories each having its own distinct (nonoverlapping) naming conventions (see section 2.10). Packages that belong to a group have names that are four to six characters in length with the first three corresponding to the name of the package group, which serves as the unit of release (UOR). Typical standalone packages have names that are seven or more characters in order to ensure that they remain disjoint from those of all grouped packages. In rare cases, particularly for very widely used (or standard) libraries, we may choose to create a package-group sized package having just a single three-character prefix, such as **`bts`** (or **`std`**). Although having a single ultra-short namespace name across a very large number of components can sometimes enhance productivity across a broad client base, such libraries typically demand significantly more skill and effort to develop and maintain than their less coarsely named package-group-based counterparts. The use of (architecturally insignificant) subpackages to support such nominally monolithic libraries is discussed in section 2.11.

[24] This nomenclature stems from way back before standardization, and we had to use logical package prefixes to implement logical namespaces — e.g., `bget_Point` instead of `bget::Point`. Even with the advent of the `namespace` construct in the C++98 Standard, we continue to exploit this approach to naming of physical entities and, occasionally, even logical ones — e.g., in procedural interfaces (see section 3.11.7).

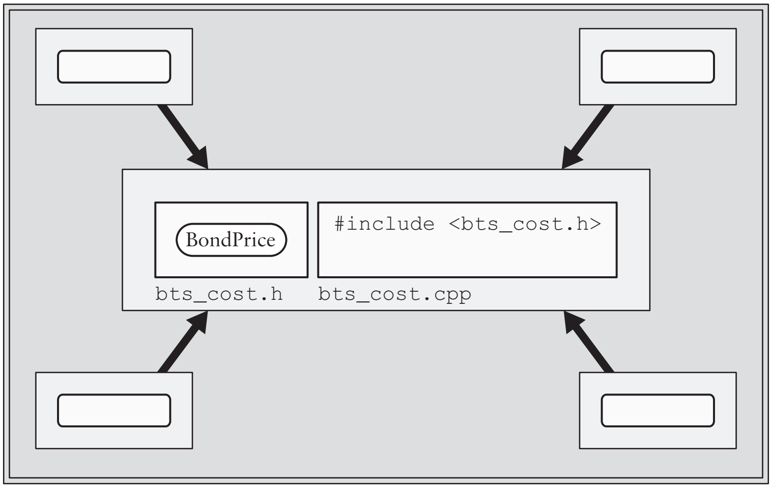


Figure 2-18: Component names always reflect their enclosing package.

Our preference that the names of physical entities (e.g., files, packages, and libraries) not contain any uppercase letters (section 1.7.1) begins with the observation that some popular file systems — Microsoft's NTFS, in particular — do not distinguish between uppercase and lowercase.[25] Theoretically, it is sufficient that the *lowercased* rendering of all filenames be unique. Practically, however, having any unnecessary extra degree of freedom in our physical packaging, thereby complicating development/deployment tools, let alone human comprehension, makes the use of mixed-case filenames for C++ source code suboptimal.[26]

Separately, and perhaps most importantly, we find that having class names, which we consistently render in mixed case (section 1.7.1) — being distinct from physical names, which we render in all lowercase — is notationally convenient and also visually reinforces the distinction

[25] With the intent of improving readability (and/or nominal cohesion), it is frequently suggested that we change to allow uppercase letters in component filenames and require them to match exactly the principal class or common prefix of contained classes (see section 2.6), instead of the *lowercased* name as is currently required. We recognize that the readability of multiword filenames can suffer (ironically providing a welcome incentive to keep component base names appropriately concise).

[26] Insisting that our component filenames be rendered in `all_lowercase` also effectively precludes "overloading" on case for logical names, e.g., having both `DateTimeMap` and `DatetimeMap` in separate components — which, from a readability standpoint, is something we would probably want to avoid anyway. Imagine trying to communicate such a distinction over a customer-service telephone hotline!

between these two distinct dimensions of design, e.g., in component/class diagrams such as the one shown above (Figure 2-18). The utility afforded by this visual distinction within source code and external documents, such as this book, should not be underestimated.

Although the `namespace` construct can and will be used effectively with respect to *logical* names, it cannot address the corresponding physical ones — i.e., component filenames. That is, even with namespaces, having a header file employing a simple name such as `date.h` is still problematic. We could, as many do, force clients to embed a partial (relative) path to the appropriate header file (e.g., `#include <bts/date.h>`) within their source code; however, ensuring enterprise-wide uniqueness in the filename itself (e.g., `#include <bts_date.h>`) provides superior flexibility with respect to deployment.[27] In other words, by making all component filenames themselves unique by design (irrespective of relative directory paths), we enable much more robustness and flexibility with respect to repackaging during deployment (see section 2.15.2).

Taking a software vendor's perspective, an early explicit requirement of our packaging methodology was the ability to select one component, or an arbitrary set of specific components, from a vast repository, extract (copies of) them along with just the components on which those components depended (directly or indirectly), and make these components available to customers as a library having a single ("flat") include directory and a single archive. Had we allowed our development directory structure to adulterate our source files, we would be forced to replicate a perhaps very large and sparsely populated directory structure on our clients' systems. Similarly, nonunique `.cpp` filenames would make re-archiving `.o` files from multiple packages into a single library archive anything but straightforward.

This unnecessarily sparse directory structure would be exacerbated by a third level of physical aggregation. For example, the same header that resided within the package-level `#include` directory during development can co-exist (i.e., within a single group-level `#include` directory) alongside headers from other packages grouped together within the same UOR, which can be more convenient (and also more efficient[28]) for use by external clients. Having this superior flexibility in deployment — especially for library software — trumps any arguments based on aesthetics or "common practice."

---

[27] We assert (see section 2.10.2) that this approach is viable for even the largest of source-code repositories. For example, see **potvin16**.

[28] **lakos96**, section 7.6.1 (pp. 514–520), and, in particular, Figures 7-21 and 7-22 (p. 519 and p. 520, respectively)

There are other collateral benefits for ensuring globally unique filenames. Having the filename embody its unique package prefix also simplifies predicting include-guard names. As illustrated in Figure 1-40, in section 1.5.2, the guard name is simply the prefix `INCLUDED_` followed by the root filename in uppercase (e.g., for file `bts_bondprice.h` the guard symbol is simply `INCLUDED_BTS_BONDPRICE`). Compilers often make use of the implementation filename as the basis for generating unique symbols within a program — e.g., for virtual tables or constructs in an unnamed namespace. Hard-coding the unique package prefix in the filename also means that its globally unique identity is preserved outside the directory structure in which it was created — e.g., in `~/tmp`, as an email attachment, or on the printer tray. Consistently repeating the filename as a comment on the very first line of each component file, as we do (see section 2.5), further reinforces its identity. Knowing the context of a file simply by looking at its name is a valuable property that one soon comes to expect and then depend on.

Design Rule

Each logical entity declared within a component must be nested within a namespace having the name of the package in which that entity resides.

Before the introduction of the `namespace` keyword into the C++ language (and currently for languages such as C that do not provide a logical namespace construct), the best solution available was to require that (where possible) the name of every logical entity declared at file scope begin with a (registered) prefix unique to the architecturally significant physically cohesive aggregate immediately enclosing them, namely, a package.[29] Attaching a logical package prefix to the name of every architecturally significant logical entity within a component, albeit aesthetically displeasing to many, was effective not only at avoiding name collisions, but also at achieving nominal cohesion, thereby reinforcing logical/physical coherence. A reimplementation of the physical module of Figure 2-17 (above) using logical package prefixes (now deprecated) is shown for reference only in Figure 2-19.

[29] **lakos96**, section 7.6.1, pp. 514–520, and in particular Figure 7-21, p. 519

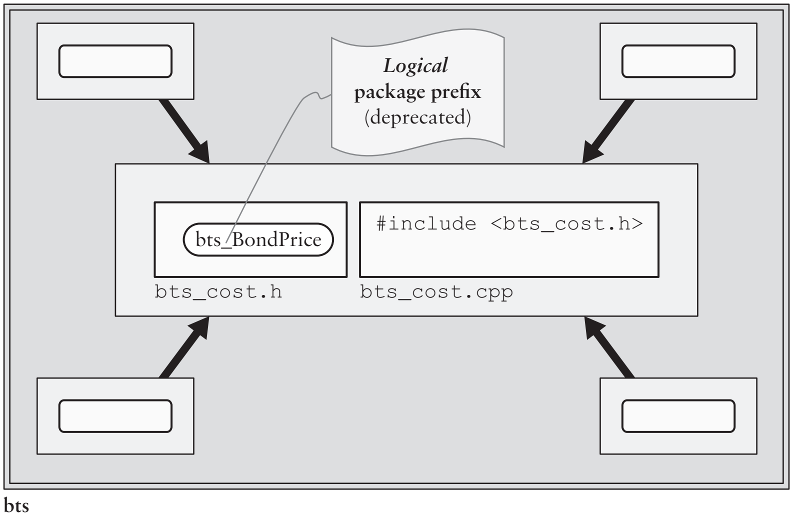


Figure 2-19: (Classical) logical package prefixes (deprecated)

Now that the `namespace` construct has long since been supported by all relevant C++ compilers, there has been an inculcation toward having concise, unadulterated logical names. Hence, we now (since c. 2005) nest each logical entity within a namespace having the same name as the package containing the component that defines the construct, as shown in Figure 2-20. Our use of logical package namespaces is isomorphic to our original use of logical package prefixes, and therefore consistent with our continued use of physical package prefixes for component filenames to preserve logical and physical name cohesion.

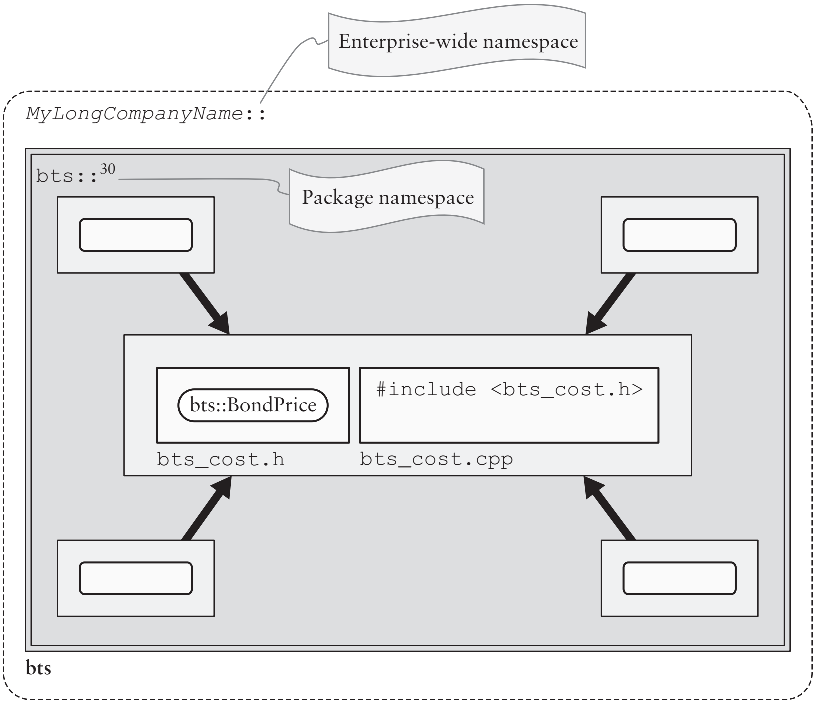


Figure 2-20: (Modern) logical package and enterprise namespaces

2.4.7 Proprietary Software Requires an Enterprise Namespace

Notice how Figure 2-20, section 2.4.6, anticipates that we now also recommend an overarching enterprise-wide namespace as a way of enabling us to disambiguate (albeit extremely rare in practice) collisions with other software that might follow our (or a similar) naming methodology.

Design Rule

Each package namespace must be nested within a unique enterprise-wide namespace.

By shielding all of our proprietary code (other than application `main` functions, see section 2.13) behind a single enterprise-wide name, e.g., our full company name (as illustrated in

[30] Note that when namespaces are not appropriate (e.g., functions having `extern "C"` linkage), we revert back to the use of logical package prefixes (see section 3.11.7).

Figure 2-20, section 2.4.6), we all but eliminate any chance of accidental external collision. And, since all of our components reside within the same enterprise namespace, there is no need or temptation to employ `using` declarations or directives.[31] In the very unlikely event that a collision with external software occurs — even in the presence of `using` directives — all that is required to disambiguate the collision is to prepend (1) the firm-wide symbol, (2) the third-party product's symbol, or (3) `::` if the third-party code failed to take this precaution.

Having, instead, each individual package represented by a namespace at the highest level would lead, at least conceptually, to myriad short global symbols, combinatorially increasing the probability of collision with vendors adopting a similar strategy (see the birthday problem in Volume III, section 8.3).[32] In any event, having a single (somehow unique) enterprise-wide "umbrella" namespace for our own code serves to mitigate risk and is therefore desirable.

The next step in achieving logical and physical name cohesion is to formalize how logical entities defined within a component are named so that their use alone identifies the component in which they are defined. To simplify the description, we provide the following definition of a component's base name.

DEFINITION: The *base name* of a component is the root name of the component's header file, excluding its package prefix and subsequent underscore.

For example, the *base name* of the component illustrated in Figure 2-20, section 2.4.6, is **`cost`**. This name, however, fails to achieve nominal cohesion with the class `BondPrice`, which it defines.

---

[31] Note that for large code bases that make significant use of templates, having a long enterprise namespace name can prove prohibitive with respect to the size of the debug symbols that the compiler generates, which may force us to go for a much shorter name — e.g., our stock ticker.

[32] Decentralized registration of packages via package groups (see section 2.9.4) is effective at managing naming conflicts within a single organization. We can, however, easily envisage a world in which source code from multiple enterprises having distinct naming regimes (consistent with our methodology) needs to co-exist within a single code base. Under those circumstances, there might be affirmative value in preventing accidental header-file collisions by proactively adding a very short (e.g., exactly *two*-character) mutually unique *physical* prefix (e.g., "`bb_`") to each organization's component names corresponding to (but not necessarily the same as) their respective unique enterprise-wide (logical) namespace names (see sections 2.4.6, 2.4.7, and 2.10.2).

2.4.8 Logical Constructs Should Be Nominally Anchored to Their Component

DEFINITION: An *aspect function* is a named (member or free) function of a given signature having ubiquitously uniform semantics (e.g., `begin` or `swap`) and, if free, behaves much like an operator — e.g., with respect to argument-dependent lookup (ADL).

Design Rule

The name of every logical construct declared at package-namespace scope — other than free *operator* and *aspect* functions (such as `operator==` and `swap`) — must have, as a prefix, the base name of the component that implements it; macro names (`ALL_UPPERCASE`), which are not scoped (lexically) by the package namespace, must incorporate, as a prefix, the entire uppercased name of the component (including the package prefix).

COROLLARY: The fully qualified name (or signature, if a function or operator) of each logical entity declared within an architecturally significant component header file must be unique throughout the enterprise.

Naming a component after its principal class or `struct` (but in all lowercase), as shown in Figure 2-21, usually resolves most potential ambiguity. For example, we would expect that class `bts::PackedCalendar` would be defined in a component called **`bts_packedcalendar`** (or conceivably, **`bts_packed`**, if the component defined other intimately related "packed" types). Note that in our methodology, however, we tend to have a single (principal) class per component unless there is one of four specific countervailing reasons to do otherwise (see section 3.3.1). Whenever there is more than one class defined at package-namespace scope within a single component, each such class name will incorporate that component's base name (albeit in "UpperCamelCase") as a prefix.[33]

[33] Note that this rule may not apply when the external ("client-facing") component headers are already specified otherwise — e.g., standardized interfaces or established legacy libraries.

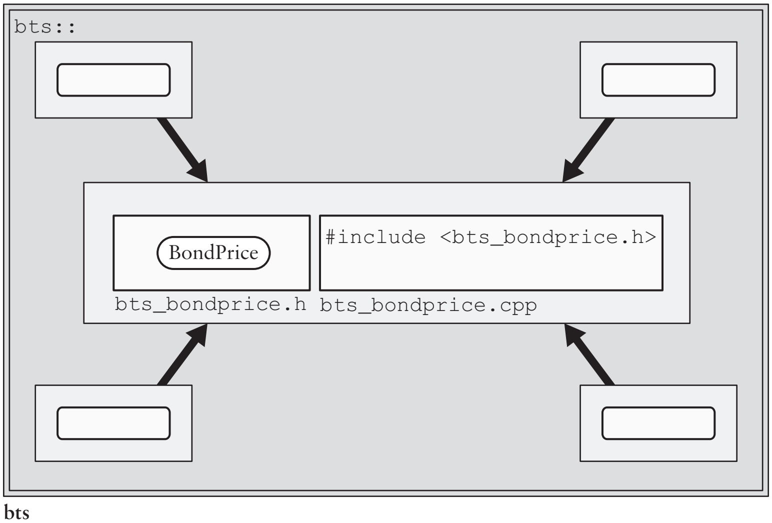


Figure 2-21: Nominally cohesive class and component (GOOD IDEA)

Where appropriate, we routinely define outwardly accessible ("public") auxiliary classes, such as iterators, in the same component either by appending to the name of the primary class (e.g., `bdlt::PackedCalendarHolidayIterator`), or else by nesting the auxiliary class within the principal class itself (e.g., `PackedCalendar::HolidayIterator`).[34] Note, however, that some detective work might be unavoidable when operators, inheritance, or user-defined conversion are involved. The rules surrounding the placement of free operators within components are discussed below.

2.4.9 Only Classes, `structs`, and Free Operators at Package-Namespace Scope

Design Rule

Only classes, `structs`, and free operator functions (and operator-like *aspect* functions, e.g., `swap`) are permitted to be *declared* at package-namespace scope in a component's `.h` file.

[34] In practice, the nested iterator type, `PackedCalendar::HolidayIterator`, would likely be a `typedef` to the non-nested auxiliary iterator class, `bts::PackedCalendarHolidayIterator`, which grants the container private (`friend`) access (e.g., see section 3.12.5.1). The mandatory colocation of two classes where one grants private access to another is discussed in section 2.6.

To minimize clutter, we have consistently avoided declaring individual functions as well as enumerations, variables, constants, etc., at namespace scope in component header files, preferring instead always to nest these logical constructs within the scope of an appropriate `class` or `struct`.[35] In so doing, we anchor these less substantial constructs within a larger, architecturally significant logical entity that, unlike a namespace (section 1.3.18), is necessarily fully contained within a single component (section 0.7). We understand that this rule, like the previous one, might not be applicable when there are valid countervailing business reasons such as an externally specified ("client-facing") interface.[36]

Having modifiable global variables at namespace scope is simply a bad idea. Nesting such variables within a class as `static` data members and providing only functional access is also generally a bad idea, but at least addresses the issue of nominal cohesion. On the other hand, nesting compile-time-initialized constants along with `typedef` declarations[37] within the scope of a class or `struct` is perfectly fine. Requiring that enumerations be nested within a class, `struct`, or function ensures that all of the enumerators are scoped locally and cannot collide with those in other components within the same package namespace.[38]

---

[35] **lakos96**, section 2.3.5, p. 77–79, in particular p. 77

[36] Sometimes it might be useful to *know* that the name of a class is itself unique throughout the enterprise. For example, if for some reason we were to implement *streaming* (a.k.a. *externalization* or *serialization*) of polymorphic objects outside of our process space (see Volume II, section 4.1), it would be important that we identify uniquely the concrete class that we are streaming. One common and effective approach is to prepend the stream data with the character string name of the concrete class whose value we are transmitting. As with the include guard symbols for files (section 1.5.2), this process is reduced to rote mechanics, provided we are assured that the name of every potentially streamable concrete class in our organization is guaranteed to be unique. Logical package prefixes (now predicated) addressed this issue directly, but we can still achieve the same effect by streaming the (ultra-concise) package name (section 2.10.1) followed by that of the class, along with a (single-character) delimiter (of course).

[37] `typedef` declarations, although often useful (e.g., to specify an *aspect*, as in `SomeContainer::iterator`), obscure the underlying types in code and, consequently, can easily detract from readability. In particular, one would not typically use a `typedef` to alias a fundamental type to one more specific to its application — e.g.,

```
typedef int NumElements;
```

would be a BAD IDEA. Separately, there would ideally be a single C++ type to represent each truly distinct *platonic* type used widely across interface boundaries (see Volume II, section 4.4).

[38] C++11 provides what is known as an `enum class`, which addresses the issue of scoping the enumerators, as well as providing for stronger type safety. Note that all enumerations in C++11 allow their underlying integral type to be specified and, unlike C++03, thereby form what is known as a *complete type*, enabling them to be declared and used locally (i.e., without also specifying the enumerators). The ability to elide enumerators can constitute what is sometimes referred to in tort law as an "attractive nuisance" in that, unless the elided enumeration is supplied by a library in a header separate from the one containing its complete definition, a client wishing to insulate itself from the enumerators would be forced to declare the enumeration locally in violation of Component Property 3 (section 1.6.3).

The justification for avoiding free functions, except operator and operator-like “aspect” functions, which might benefit from argument-dependent lookup (ADL), derives from our desire to encapsulate an appropriate amount of logically and physically coherent functionality within a nominally cohesive component. While classes are substantial architectural entities that are easily identifiable from their names, individual functions are generally too small and specific for each to be made nominally cohesive with the single component that defines them, as in Figure 2-22a.[39]

Creating components that hold multiple functions in which there is no nominal cohesion (Figure 2-22b) makes human reasoning about such physical nodes much more difficult and is therefore also a bad idea. Forcing the name of each function to have, as a prefix, the initial-lowercased rendering of the base name of the component (Figure 2-22c) achieves nominal cohesion, but is awkward at best, and fails to emphasize logical coherence (section 2.3). We could employ a third level of namespace (Figure 2-22d), but for reasons discussed below (Figure 2-23) and also near the end of section 2.5, we feel that would be suboptimal.

```
// xyza_roundtowardzero.h

namespace xyza {

double roundTowardZero(double value);

}  // close package namespace
```

(a) Nominally cohesive function at package-namespace scope (BAD IDEA)

```
// xyza_mathutil.h

namespace xyza {

double roundTowardZero(double value);

double factorial(double value);

}  // close package namespace
```

(b) Nominally noncohesive functions at package-namespace scope (BAD IDEA)

[39] Given that we virtually always open and close a package namespace exactly once within a component (see section 2.5), we choose not to indent its contents, thereby increasing usable real estate given a practical maximum line length (e.g., 79) suitable for efficient reading, printing, side-by-side comparison, etc. (see Volume II, section 6.15).

```
// xyza_mathutil.h

namespace xyza {

double mathUtilRoundTowardZero(double value);

double mathUtilFactorial(double value);

}  // close package namespace
```

(c) Nominally cohesive functions at package-namespace scope (AWKWARD)

```
// xyza_mathutil.h

namespace xyza {

namespace MathUtil {

    double roundTowardZero(double value);

    double factorial(double value);

}  // close local namespace

}  // close package namespace
```

(d) Nominally cohesive namespace containing functions (NOT OPTIMAL)

```
// xyza_mathutil.h

namespace xyza {

struct MathUtil {

    static double roundTowardZero(double value);

    static double factorial(double value);

};

}  // close package namespace
```

(e) Nominally cohesive utility `struct` containing functions (WHAT WE DO)

Figure 2-22: Ensuring nominal cohesion for free functions and components

We therefore generally avoid declaring free *(nonoperator)* functions at package-namespace scope, and instead achieve both nominal logical and physical cohesion by grouping related functionality within an extra level of namespace matching the component name using `static` methods within a `struct` (Figure 2-22e), which we will consistently refer to as a *utility* (see section 3.2.7) and so indicate with a `Util` suffix (e.g., `xyza::MathUtil`).[40] Additional, collateral advantages for preferring a `struct` (e.g., Figure 2-22e) over a third level of `namespace` (e.g., Figure 2-22d) for implementing a *utility* are summarized in Figure 2-23.[41]

There are many advantages of using a `struct` (e.g., Figure 2-22e) over a third level of namespace (e.g., Figure 2-22d) for aggregating related (what would otherwise be *free*) functions into a single *utility* component.

(1) The distinct syntax and atomic nature of a `struct` having `static` methods makes its purpose as a component-scoped entity clearer than would yet another, nested `namespace`, leaving namespaces for routine use at the package and enterprise levels exclusively.

(2) The self-declaring nature of functions and data defined at namespace scope (section 1.3.1) are necessarily eliminated when they are instead nested (as `static` members) within a `struct`.

(3) Unlike a `namespace`, a `struct` does not permit `using` directives (or declarations) to import function names into the current (e.g., package) namespace, thereby preventing any consequent loss in readability.[42]

(4) Unlike a `namespace`, a `struct` can support private nested data — e.g., as an optimization for accessing *insulated* (external bindage) table-based implementation details, residing in the `.cpp` file, by one or more inline functions, residing in the `.h` file (see Volume II, section 6.7).

(5) Unlike a `namespace`, a `struct` can be passed as a template parameter — e.g., as a cartridge of related functions satisfying a concept (e.g., see Figure 3-29, section 3.3.7).

(6) Unlike a `namespace`, a C-style function in a `struct` does not participate in Argument-Dependent Lookup (ADL), thereby avoiding potentially large overload sets, which could needlessly affect compile-time performance and possibly introduce unanticipated (perhaps even latent) ambiguity, or — much worse — invoke the wrong function.[43] By placing our "free" functions in a `struct`, we make our design decision not to employ ADL explicit.

(7) Except for a few very stylized cases, such as `std::placeholders` (e.g., _1, _2, _3) and `std::literals`, use of namespace declarations are generally ill-advised. Should we subsequently discover a rare valid engineering reason for enabling local `using` declarations, we can easily migrate a `struct` to a `namespace` by creating a new component-private `struct` (see section 2.9.1), e.g., `MathUtil_Imp`, and forwarding calls to it from the new nested (e.g., `MathUtil`) namespace. Note that, except when used as in (5), it is always possible to migrate from a `struct` to a `namespace` without forcing any clients to rework their source code, but, given the possibility of `using` directives/declarations, not vice versa (see Volume II, section 5.5).

Figure 2-23: Prefer `struct` to `namespace` for aggregating "free" functions.

---

[40] Note that it is not possible to have partial specializations for static method templates in a `struct` the way you can for free-function templates.

[41] Because only free (i.e., non-member) functions participate in ADL, extending the C++ language to accommodate new features, e.g., redeclaration (**voutilainen19**), for such functions (as opposed to `static` members of a `struct`) is considered by some to be substantially more technically difficult to implement in relevant C++ compilers. For more on why such extensions might be practicably useful in future incarnations of the C++ language, see Volume II, section 6.8.

[42] Although `using` declarations can be used to import declarations of overloaded functions of a given name from a private (or protected) base class into a public one, we generally discourage such use, as it would require a public client to view otherwise private (or protected) detail; instead, we prefer to create (and document) an inline forwarding function. Note that a similar issue arises with forwarding constructors as of C++11.

[43] Titus Winters of Google has recently (c. 2018) expressed increasing concerns as to the scalability and stability of such overload sets (**winters18a**, "ADL"); see also **winters18b**, particularly starting at the 11:30 time marker.

Design Rule

A component header is permitted to contain the declaration of a *free* (i.e., non-member) operator or *aspect* function (at package-namespace scope) only when one or more of its parameters incorporates a type defined in the same component.

In our methodology, operators, whether member or free, are by their nature fundamental to the type(s) on which they operate. Every unary and homogeneous binary operator — i.e., one written in terms of a single user-defined type, e.g.,

```
bool operator==(const BondPrice& lhs, const BondPrice& rhs);
```

is declared and defined within the same component (e.g., **`bts_bondprice`**) as the type (e.g., `bts::BondPrice`) on which it operates. Note that, except for forms of assignment (e.g., `=`, `+=`, `*=`), we will always choose to make a binary operator free (as opposed to a member) to ensure symmetry with respect to user-defined conversions (see Volume II, section 6.13). For conventionally heterogeneous operators such as

```
std::ostream& operator<<(std::ostream&    stream,
                         const BondPrice& price);
```

the motivation to make them free is born of extensibility without modification, as in the open-closed principle (section 0.5). In any event, the place to look for the definition of an operator (entirely consistent with ADL) is within a component that defines a type on which that operator operates.

If we were to allow free operators to be defined in arbitrary components, how could we even know if they exist? If we saw one being used, how would we track down its definition? Even more insidious is the possibility that a client unwittingly duplicates such a definition locally. The resulting latent incompatibilities, manifested by future multiply-defined-symbol linker errors, would threaten to destabilize our development process.

As an important, relevant example, consider the standard template container class, `std::vector`, for which no standard output operator is defined. Referring to Figure 2-24, suppose that the author of component **`my_stuff`** finds outputting a vector to be generally useful, and so "thoughtfully" provides

```
template <class TYPE>
std::ostream& operator<<(std::ostream&            lhs,
                         const std::vector<TYPE>& rhs);
```

(along with an appropriate definition) in its header for general use by clients. It is not hard to imagine that component **your_stuff** might do so as well. Now consider what happens when their_stuff.cpp includes both my_stuff.h and your_stuff.h. The inevitable result is multiply defined symbols![44]

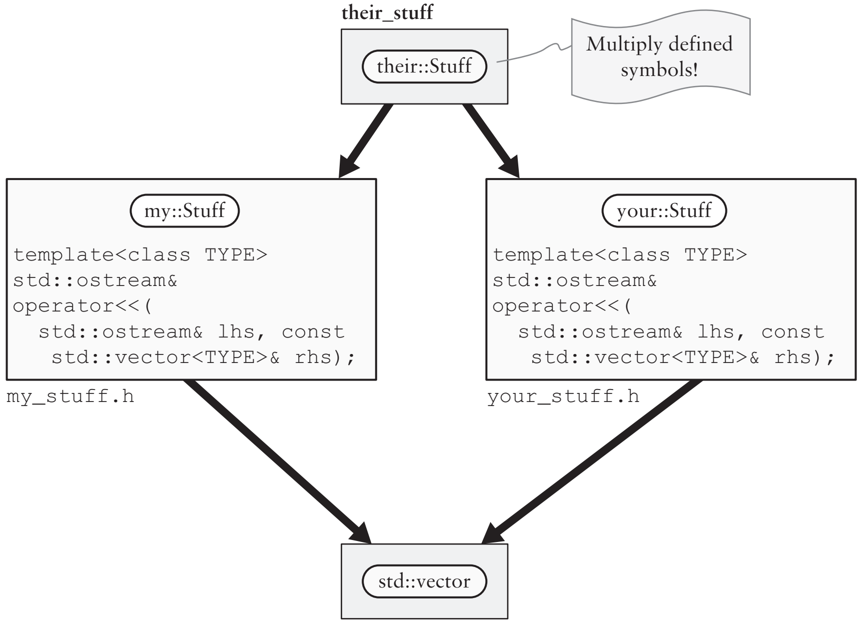


Figure 2-24: Problems with defining operators in unexpected components

Instead, the functionality should have been implemented as a static member function of a utility struct (see section 3.2.7) in a separate component, as illustrated in Figure 2-25.

[44] Because the offending operator is a template, which has dual bindage (section 1.3.4), it is entirely possible that the duplicate definitions will go unnoticed by either the compiler or the linker for quite some time — that is, until the compiler can see the two template definitions side-by-side in a single translation unit. Had the construct instead had external bindage, such as an ordinary function or an explicit instantiation, merely linking the two components into the same program would have been sufficient to expose the incompatibility.

```
// xyza_printutil.h

// ...

namespace xyza {

// ...

struct PrintUtil {

    // ...

    template<class TYPE>
    static std::ostream& print(std::ostream&      stream,
                               const std::vector<TYPE>& object);

    // ...
};

// ...

}  // close package namespace

// ...
```

Figure 2-25: Avoiding free operators on nonlocal types

As illustrated in Figure 2-26, providing an output operator on a type `my::Type` — or conceivably even on a `std::vector<my::Type>` — in component **`my_type`** is perfectly fine. The general design concept being illustrated here is to follow the teachings of the philosopher Immanuel Kant and avoid doing those things that, if also done by others, would adversely affect society (see section 3.9.1). By adhering to this simple rule for operators, we ensure that (1) we know where to look for each operator, and (2) operator definitions will not be duplicated (and therefore cannot conflict at higher levels in the physical hierarchy).

```
// my_type.h
// ...

namespace my {

class Type {
    // ...
};

std::ostream& operator<<(std::ostream& stream, const Type& object);   Correct

std::ostream& operator<<(std::ostream&            stream,             Not wrong
                         const std::vector<Type>& object);

}  // close package namespace

// ...
```

Figure 2-26: Overloading free operators on types within the same component

If a single free operator refers to two types implemented in separate components, where one depends on the other, the operator would of course be defined in the higher-level component. If, however, the components are otherwise independent (as illustrated Figure 2-27a), we have two alternatives:

1. [Suboptimal] Arbitrarily choose one of the components to be at a higher-level and place the free operator there, as in Figure 2-27b (thus introducing additional physical dependency for one of the components).
2. [Preferred] Create a utility class in a separate component, as in Figure 2-27c, and define one or more nonoperator functions nested within a `struct` that serves the same purpose (see section 3.2.7). Note that it is *never* appropriate to *escalate* (see section 3.5.2) co-dependent free operators to a separate component.

Use of operators for anything but the most fundamental, obvious, and intuitive operations (see Volume II, section 6.11) are almost always a bad idea and should generally be avoided; any valid, practical need for operators across otherwise independent user-defined types is virtually nonexistent.[45]

[45] We note that the C++ streaming operators and Boost.Spirit are (rare) arguably plausible counter-examples; still, we maintain that heterogeneous equality comparison operators across disparate user-defined value types (see Volume II, section 4.1), such as `Square` and `Rectangle` (Figure 2-27), remain invariably misguided for entirely different reasons (see Volume II, section 4.3).

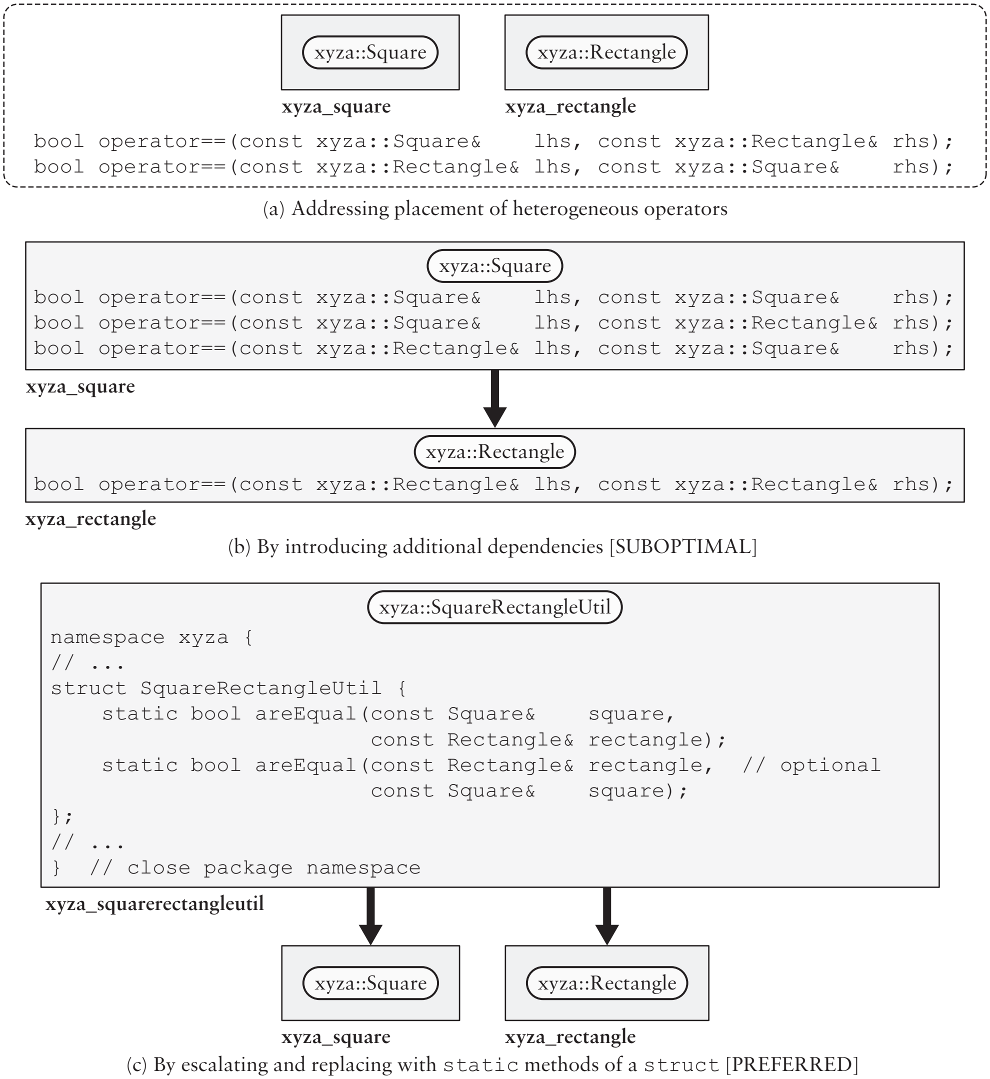


Figure 2-27: Implementing "free operators" referring to multiple peer types

2.4.10 Package Prefixes Are Not Just Style

Make no mistake, how packages are named is not just a matter of style; package names have profound architectural significance. As an example, consider Figure 2-28, which shows a hierarchy of components whose dependencies form a binary tree. Clearly these components are levelizable (section 1.10) and, hence, have no cycles. However, it is not in general possible to assign components of a multipackage subsystem to arbitrary packages without introducing package-level cycles. In this example, the packages containing these components (as implied by the package prefixes embedded in the component names) would be cyclic and therefore *not* levelizable.

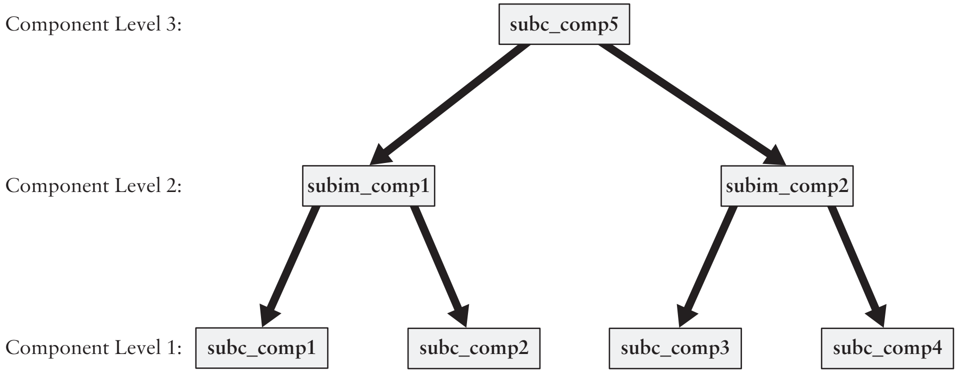


Figure 2-28: Implied cyclic package dependencies (BAD IDEA)

The problem, identified by Figure 2-29, can easily arise in practice. Consider the design of a single package that is intended to contain everything that is directly usable by clients of a multipackage subsystem. If this presentation package (**`subc`**) defines both protocol (i.e., pure abstract interface) classes (which are inherently very low level) and wrapper components (which are inherently very high level), it will not be possible to interleave components from a separate implementation package (**`subim`**).[46]

---

[46] For complex subsystems, the implementation components represented here as a single package **`subim`** may appropriately span many packages at several different levels; however, the basic idea remains the same.

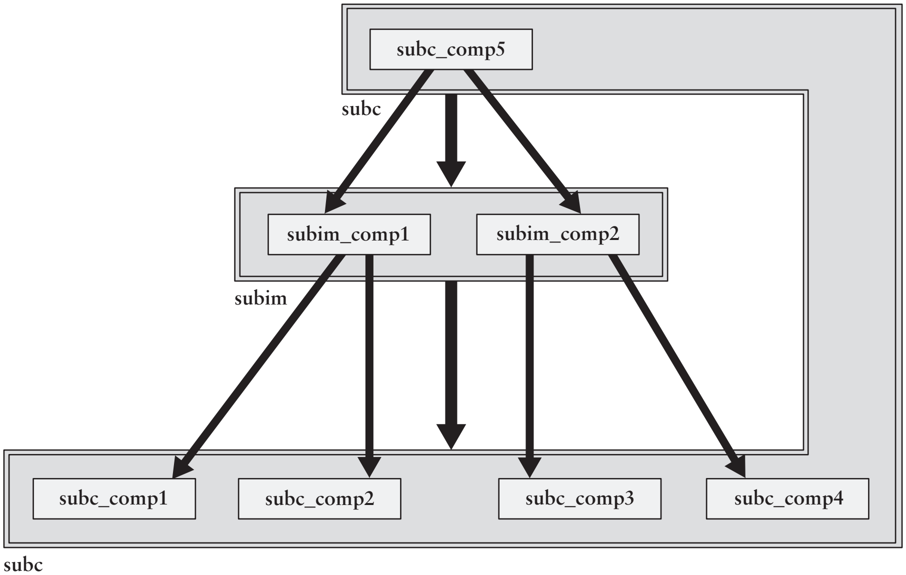


Figure 2-29: Acyclic component hierarchy; cyclic package hierarchy (BAD IDEA)

> **COROLLARY: Allowed (explicitly stated) dependencies among packages must be acyclic.**

Allowing cyclic dependencies among packages, like any other aggregate, would make our software qualitatively more complicated. Ultimately, *all* cyclically involved packages would have to be treated as a unit. A general solution to this common problem, illustrated in Figure 2-30, is simply to provide two separate client-facing packages. One package (**`subw`**) will reside at the top of the subsystem and contain components that define only wrappers[47] (e.g., **`subw_comp1`**); the second will reside at the bottom of the package hierarchy and incorporate components

---

[47] A *wrapper* is a *facade* that allows clients to manipulate objects (typically of some other type) without providing direct programmatic access to those objects (see sections 3.1.10 and 3.11.6).

(e.g., **subv_comp1**) that define protocol and other *vocabulary* types (see Volume II, section 4.4) exposed programmatically through the wrapper interface.[48]

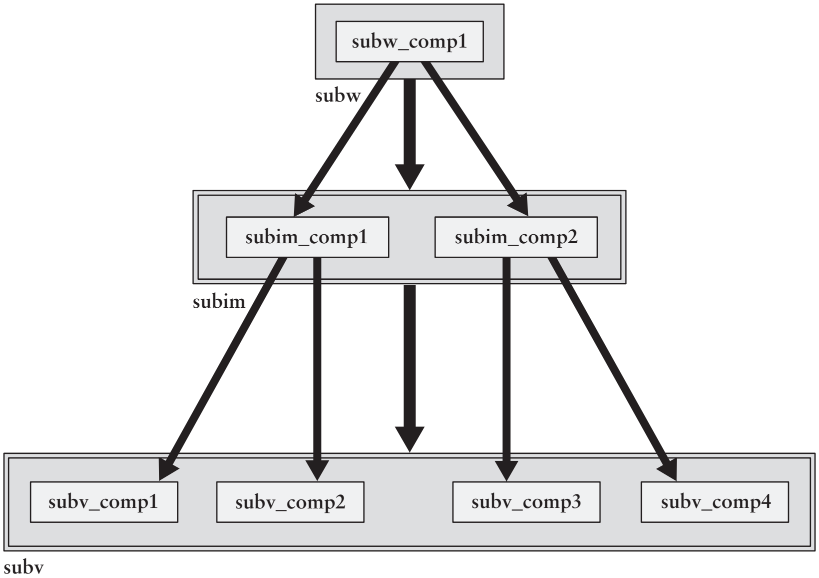
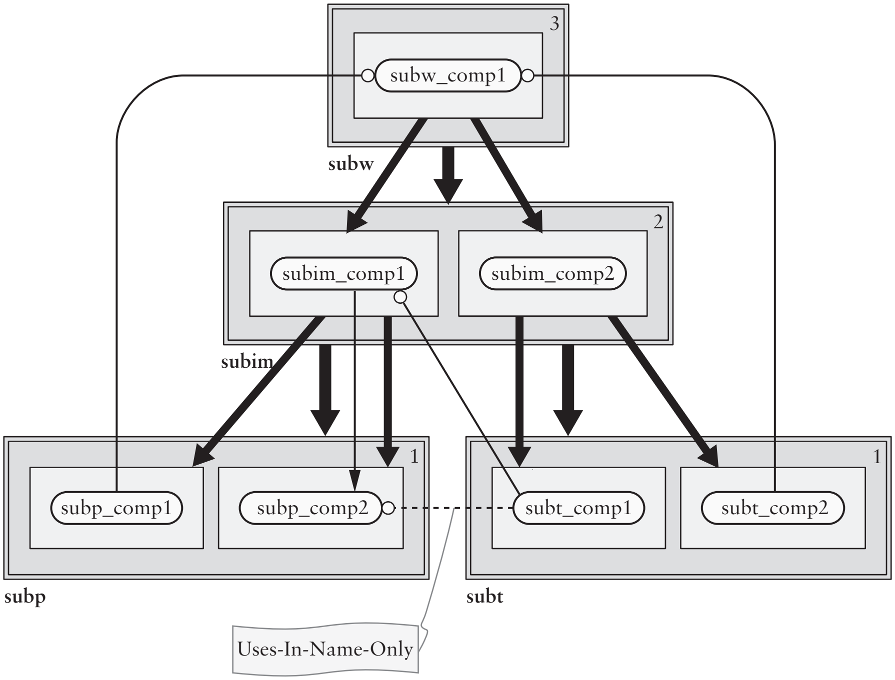


Figure 2-30: Repackaging of components to avoid cyclic package dependencies

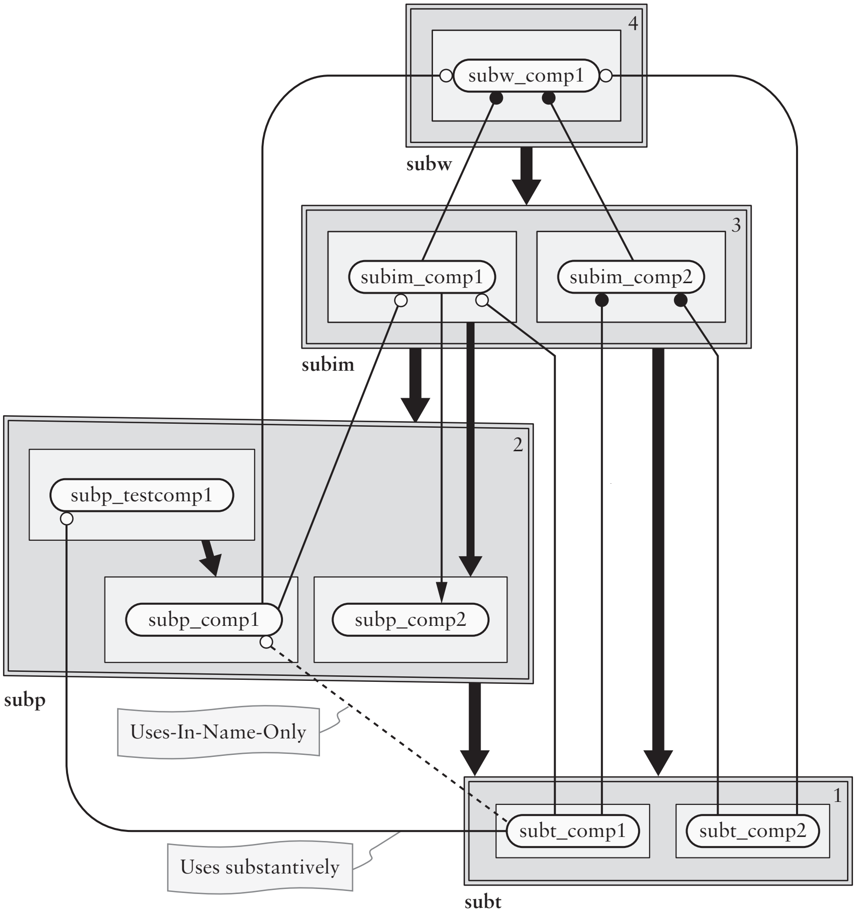
Components that are used in the interface of the wrapper components (**subw**), and also *in name only* by low-level protocols, typically reside either in the same package as the protocols (e.g., **subv** in Figure 2-30) or in a separate, lower-level package, as illustrated in Figure 2-31b, as opposed to at the same level (Figure 2-31a), in order to enable concrete test implementations of the protocols to properly reside along with them (e.g., in **subp**), yet allow such test implementations to depend on the actual concrete vocabulary types (e.g., in **subt**) rather than having to mock them.

---

[48] See the *escalating encapsulation* levelization technique (section 3.5.10).



(a) Parallel protocol and concrete vocabulary-type packages (BAD IDEA)



(b) Subordinate local vocabulary-type package (GOOD IDEA)

Figure 2-31: Alternative packaging strategies

2.4.11 Package Prefixes Are How We Name Package Groups

Although packages, being architecturally significant aggregates, have unique names (and namespaces), it is often advantageous to bundle packages having similar purposes and/or similar envelopes of physical dependency into a larger, logically and physically coherent, nominally cohesive aggregate. We could make a big deal about this issue (and perhaps we should, given its importance). Instead we will avoid the drama and just make our point: The first three letters of a package name identify the physically cohesive package group in which a grouped package resides.

The reason for this simple approach is, well, simple (see section 2.10.1): We simply must have an ultra-efficient way to specify the package group and package of each component and class in order to obviate noisome and debilitating `using` directives and declarations (see section 2.4.12). The choice of three letters (as opposed to, say, two or four) is simply an engineering trade-off. This simple, concise, and effective approach to naming package groups is illustrated in Figure 2-32. We will revisit our package-naming rules (in much greater depth) in section 2.10.

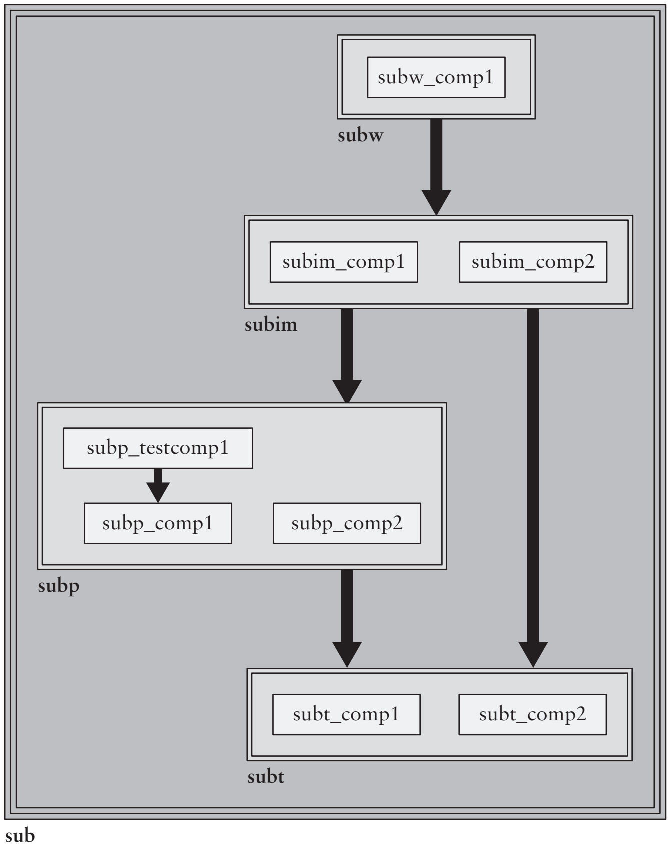


Figure 2-32: Logically and physically cohesive package group

2.4.12 `using` Directives and Declarations Are Generally a BAD IDEA

Let us now take a closer look at our use of the C++ `namespace` construct to partition logical entities along package boundaries. One of the solid benefits of package namespaces is that access to other entities local to that package does not require explicit qualification. This advantage is particularly pronounced at the application level, where much of the code that interoperates is defined locally (see section 2.13). Absent `using` directives and declarations, an unqualified reference is as informative as a qualified one: An unqualified reference implies that the entity is local to *this* package.[49]

In the code example of Figure 2-33, we cannot simply look at the definition of the `insertAfterLink` helper function and know which `Link` class we are talking about without potentially having to scan back through the entire file for preceding occurrences of `using`.

[49] There is still, however, one pragmatic reason to prefer the inflexibility of the hard-coded logical package prefix that continues to give us pause even though we have fully embraced package namespaces in our day-to-day work. Unfortunately, any use of `using` directives and declarations render case-by-case explicit use of the package namespace "tag" for remotely defined types optional, at the expense of nominal cohesion. Occasionally, library developers will need to "search the universe" for all uses of some class or utility. When we consider the possible use of `using` directives and declarations, any hope of relying on a simple search and replace (e.g., in the event a component "moves" from one package to another) is lost. Instead, we are forced to parse every line of source code. Even when we have such an elaborate tool (e.g., Clang), it, like the compiler itself, runs many orders of magnitude slower than a simple search engine looking for a fixed identifier string. We saw this same kind of speed issue with respect to determining the envelope of direct physical dependencies by scanning for just the `#include` directives nested within a component (section 1.11). Hence, use of the `namespace` construct, at least in this particular respect, is not *as* scalable as the classical, albeit archaic (and now deprecated), logical package prefix.

```
// my_link.cpp
#include <my_link.h>

// ...

#include <your_list.h>  // defines class 'Link'

// ...

namespace Foo {
    class Link { /*...*/ };  // another definition of 'Link'
}

// ...
// ...
// ...
// ...
// ...

inline
static void insertAfterLink(Link *node, Link *newNode)
{
    BSLS_ASSERT(node);
    BSLS_ASSERT(newNode);

    newNode->next = node->next;
    newNode->prev = node;
    node->next = newNode;

    if (newNode->next) {
        newNode->next->prev = newNode;
    }
}

// ...
```

Cannot determine which `Link` is being used without looking at prior `using` directives

(See Volume II, section 6.8.)

Figure 2-33: Nonlocal namespace names are optional! (BAD IDEA)

What's worse, it might be that `using` directives or declarations are not even local to the implementation file, but are instead imported quietly in one or more of many included header files as illustrated in Figure 2-34. And, unlike the C++ Standard Library (or `std` in code), which is comparatively small, unchanging, and well known, we cannot be expected to know every class within every component of every package throughout our enterprise. Still worse, nesting a variety of `using` directives and declarations within header files risks making relevant the relative order in which these headers are incorporated into a translation unit![50]

[50] **sutter05**, item 59, pp. 108–110

```
// my_app.cpp
#include <my_app.h>
#include <cdel_log.h>
#include <ddet_swap.h>
#include <ddet_table.h>
#include <ddeu_isma30360.h>
#include <dteal_technology.h>
#include <emeg_protocol.h>
#include <emem_list.h>
#include <etef_fizzbin.h>
#include <etet_trade.h>
#include <eteu_semiannual.h>
#include <fmeec_transport.h>
#include <fteem_balloon.h>
#include <ftet_account.h>
#include <ftet_position.h>
#include <ftex_prepayment.h>
// ...
// ...
// ...
#include <pcst_client.h>
#include <otem_config.h>
#include <tdep_render.h>
#include <ynot_evenmore.h>

// ...
// ...
// ...
// ...
// ...
// ...
// ...

static void communicate(Relay *relay)
{
    static Callback myCallback;

    if (relay->isOperational()) {
        relay->setForwardCallback(&myCallback):
    }
    else {
        Log::singleton().write("Life is like a box of chocolates...");
    }

    // ...
}

// ...
```

Cannot determine which `Relay` is being used even after looking at every statement in this file — `using` directives/declarations or otherwise!

Figure 2-34: `using` directives/declarations can be included! (BAD IDEA)

Design Rule

Neither `using` directives nor `using` declarations are permitted to appear outside function scope within a component.

No matter what, we must forbid any `using` directives or declarations in header files outside of function scope.[51,52,53,54] Perhaps some advocates of `using` in headers might not yet have realized that the incorporation of names from one namespace, `A`, into another, `B`, does not end with the closing brace of `B` into which names from `A` were imported, but remain in `B` until the end of the translation unit. Consequently, `using` directives or declarations are sometimes used (we should say horribly misused) in header files when declaring class member data and function prototypes to shorten the names of types declared in distant namespaces

[51] And, in library code, `using` is generally best avoided altogether. If used there at all, a `using` *declaration* (not *directive*) — whether employed to enable ADL (e.g., for a free *aspect* function, such as `swap`), or merely as a compact alias (e.g., as an entry into a dispatch table) — should appear only within a very limited lexical context, i.e., function (or block) scope.

[52] In C++98, `using` declarations replaced *access declarations* (which were deprecated intermediately and, in C++11, finally removed) for the purpose of promoting all overloads of a given (named) member function from a base class into the current scope while potentially increasing its level of access, e.g., from private to public. As we will discuss shortly, we avoid any use of class-scope `using` declarations, especially those that might force public clients to refer to less-than-public regions of a class's implementation.

[53] C++11 introduced other contexts in which the `using` keyword is valid (e.g., as an *alias declaration* used to replace `typedef`) having nothing to do with either `using` declarations or `using` directives.

[54] Alisdair Meredith notes (via personal email, 2018) that, when a base class is a template, the set of overloads to forward is an open set. Accidental breakage can occur when a design requires that each of the overloads be exposed manually. When the intent is to *perfectly forward* an overload set from a base class, a `using` declaration is a clear statement of that design intent.

Nonetheless, our recommended approach is to avoid such uses of (typically *structural*) inheritance (see Volume II, section 4.6), preferring the more compositional *Has-A* (section 1.7.4) approach to *layering* (see section 3.7.2) instead.

That said, exceptional cases do exist. Alisdair Meredith further points out (again, via personal email, 2018) that we ourselves have, on occasion, been known to introduce a base class having fewer template parameters, and then use *structural* inheritance and `using` declarations to expose that functionality as the public interface. If we were now to replace `using` declarations with, say, `inline` forwarding functions, we would negate the intended effect of reducing template-induced code bloat (see Volume II, section 4.5).

(BAD IDEA).[55] Instead, we must use the package-qualified name of each logical entity not local to the enclosing package. For this reason, we will want to ensure that widely used ("package") namespace names, like `std`, are very short indeed.

The use of `using` declarations for function forwarding during private (never mind protected) inheritance is also to be avoided because (1) our ability to document and understand such functionality in the derived header itself is compromised, and (2) inheritance necessarily implies compile-time coupling (section 1.9; see also section 3.10). We generally prefer to avoid private inheritance, in favor of layering (a.k.a. *composition*), and explicit (`inline`) function forwarding.

Finally, using namespaces to define a logical "location" independent of its physical location, say, to avoid changing `#include` directives (should some class be logically "repackaged") is — in our view — misguided. If we change the *logical* location of a class then — in our methodology — that class must be moved to its proper *physical* location as well. Unless logical and physical locations coincide, many of the advantages of sound physical design — e.g., reduced compile time, link time, and executable size (not to mention organization and understandability) — are compromised.

Adhering to these cohesive naming rules does, however, impose some extra burden on library developers. That is, if a logical construct were to "move" from one *architectural location* to another, its address (i.e., its component name), and therefore some aspect of its fully qualified logical name, *must* necessarily change as well. This "deficiency" is actually a feature in that it allows for a reasonable deprecation strategy: During refactoring, it is possible for two versions

---

[55] Local `typedef`s have historically been effective at addressing long names in data definitions and function prototypes due to specific template instantiations:

```
class Book {
    // ...
    typedef std::map<std::string, std::string>       StrStrMap;
    typedef std::map<std::string, std::vector<int> > StrIntarrayMap;
    // ...
    StrStrMap      d_glossary;
    StrIntarrayMap d_index;
    // ...
};
```

We recognize that C++11 offers `using` as a syntactic alternative, and that thoughtful (discriminating) use of `auto` can also help eliminate redundant (or otherwise superfluous) explicit type information in source code. See **lakos21**.

of the same logical entity to co-exist for a period of time as clients rework their code to refer to the new component before the original one is finally removed.[56]

2.4.13 Section Summary

In summary, our rigorous approach to cohesive naming — packages, components, classes, and free (operator) functions — not only avoids collisions, it also provides valuable visual cues within the source code that serve to identify the physical location of all architecturally significant entities. Experience shows that human cognition is facilitated by such visual associations. In turn, this nominal cohesion reinforces the even more critical requirement of logical/physical coherence (section 2.3). Hence, logical and physical name cohesion across related architecturally significant entities is an integral part of our component-based packaging methodology.

Index

D

E

I

J–K

L

V

W

X-Y-Z