Large-Scale C++

Volume I

Process and Architecture

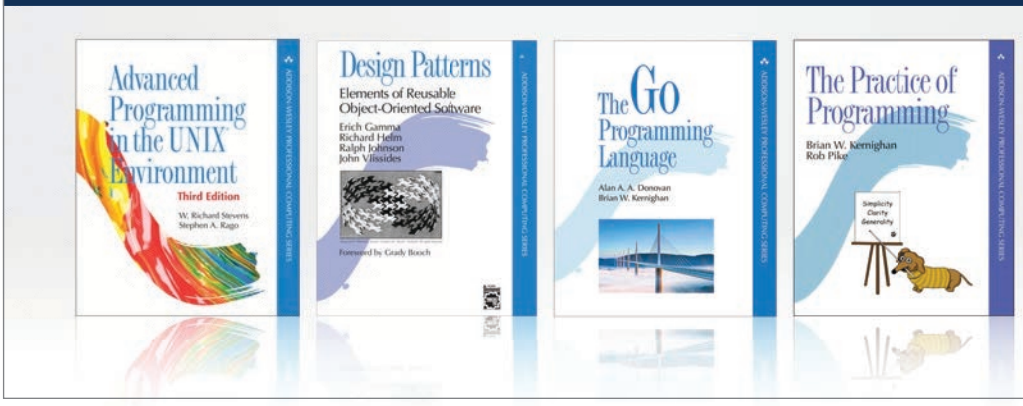
John Lakos

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Large-Scale C++

The Pearson Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor



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

The **Pearson Addison-Wesley Professional Computing Series** was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. Pearson Addison-Wesley is renowned for publishing accurate and authoritative books on current and cutting-edge technology, and the titles in this series will help you understand the state of the art in programming languages, operating systems, and networks.

Make sure to connect with us!
informit.com/socialconnect

Pearson
Addison-Wesley

informIT.com
the trusted technology learning source

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

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

This page intentionally left blank

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.

This page intentionally left blank

0

Motivation

Large-scale, highly maintainable software systems don't just happen, nor do techniques used successfully by individual application developers necessarily scale to larger, more integrated development efforts. This is an *engineering* book about developing software on a large scale. But it's more than just that. At its heart, this book teaches a skill. A skill that applies to software of all kinds and sizes. A skill that, once learned, becomes almost second nature, requiring little if any additional time or effort. A skill that repeatedly results in organized systems that are fundamentally easy to understand, verify, and maintain.

The software development landscape has changed significantly since my first book.[1] During that time, the Standard Template Library (STL) was adopted as part of the initial C++98 Language Standard and has since been expanded significantly. All relevant compilers now fully support exceptions, namespaces, member templates, etc. The Internet has made open-source libraries far more accessible. Thread, exception, and alias safety have become common design considerations. Also, many more people now appreciate the critical importance of sound *physical design* (see Figure 0-32, section 0.6) — a dimension of software engineering I introduced in my first book. Although fundamental physical design concepts remain the same, there are important new ways to apply them.

This book was written with the practitioner in mind. The focus is closely tied to a sequential development methodology. We describe in considerable detail how to develop software in terms of the well-defined atomic physical modules that we call *components*. A rich lexicon has been assembled to characterize the process. Many existing engineering techniques have been updated and refined. In particular, we present (see Volume III) a comprehensive treatment of component-level testing. What used to be considered a black art, or at least a highly specialized craft, has emerged into a predictable, teachable engineering discipline. We also discuss (see Volume II) the motivations behind and effective use of many essential "battle-hardened" design and implementation techniques. Overall, the engineering processes described here (Volume I) complement, and are synergistic with, proven project-management processes.

Bottom line: This book is designed for professional software developers and is all about being successful at developing software that can scale to arbitrary size. We have delineated the issues that we deem integral and present them in an order that roughly corresponds to our software-development thought process. Many important new ideas are presented that reflect a sometimes harsh reality. The value of this book, however, is not just in the ideas it contains but in the cohesive regularity with which it teaches sound engineering practices. Not everything we talk about in this book is popular (yet), but initially neither was the notion of *physical design*.

---

[1] **lakos96**

0.1 The Goal: Faster, Better, Cheaper!

The criterion for successful software application development in industry is invariably the delivery of the best product at the lowest possible cost as quickly as possible. Implicit in this goal are three fundamental dimensions:

- *Schedule (faster)*: Expediency of delivery of the software
- *Product (better)*: Enhanced functionality/quality of the software
- *Budget (cheaper)*: Economy of production of the software

In practice, we may optimize the development of a particular software application or product for at most two of these parameters; the third will be dictated. Figure 0-1 illustrates the interdependence of these three dimensions.[2]

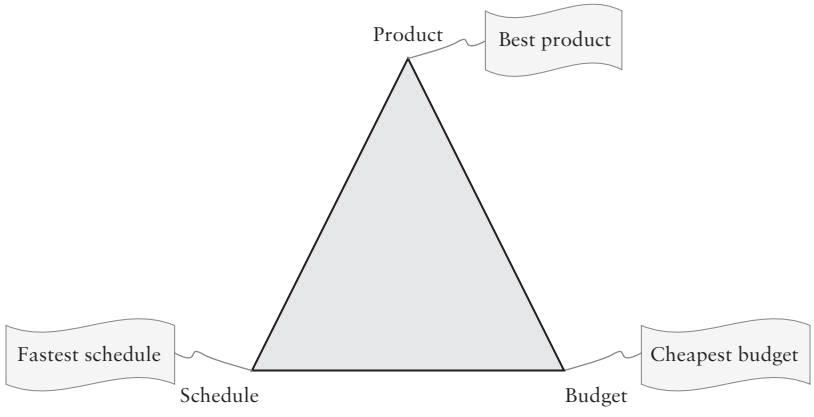


Figure 0-1: Schedule/product/budget trade-offs

At any given point, our ability to develop applications is governed by our existing infrastructure, such as developers, libraries, tools, and so on. The higher the quality goal of our product, the more calendar time and/or engineering resources it will consume. If we try to make a product of similar quality take less time and thereby improve the schedule, it will cost more — often a *lot* more, thereby negatively impacting the budget. If we have a fixed budget, the only way

[2] **mcconnell96**, section 6.6, "Schedule, Cost, and Product Trade-Offs," Figure 6-10, p. 126

to get the work done quicker is to do less (e.g., fewer features, less testing). This inescapable reality seems intrinsic to all software development.[3]

Still, it would be nice if there were some predictable way that, over time, we could improve all three of these parameters at once — that is, devise a methodology that, as a byproduct of its use, would continually reduce both cost and time to market while improving quality for future products. In graphical terms, this methodology would shift the faster/better/cheaper design space for applications and products further and further from the origin, as illustrated in Figure 0-2.

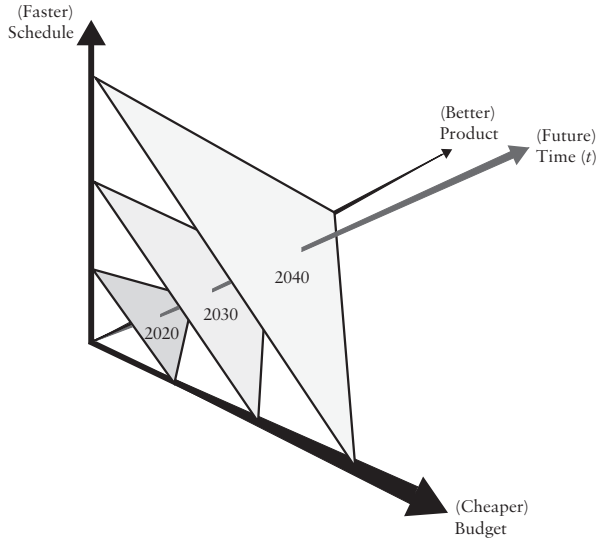


Figure 0-2: Improving the schedule/product/budget design space

---

[3] JC van Winkel has commented that these relationships are difficult to appreciate as a single graph and suggests that there are other, more intuitive ways to approach understanding these trade-offs, e.g., using *sliders*.

For a fixed schedule (calendar time to delivery), you get this slider:

(Cheaper) Budget ——●—— (Better) Product

For a fixed budget (money/resources), you get this slider:

(Better) Product ——●—— (Faster) Schedule

For a fixed product (features/quality), you get this slider:

(Faster) Schedule ——●—— (Cheaper) Budget

This final slider is at the heart of the titular thesis of Fred Brooks's classic work *The Mythical Man Month* (see **brooks75)**, which asserts that the idea that there is an inverse *linear* proportionality between *time* and *cost* that holds over the entire range of interest is pure fantasy. The geometric growth of interpersonal interactions (corroborated by empirical data; **boehm81**, section 5.3, pp. 61–64) suggests that — within a narrow band of relevant limits — this relationship might reasonably be modeled as an inverse *quadratic* one between time (T) and cost (C), i.e., $T \propto 1/\sqrt{C}$ (see section 0.9).

Assuming such a methodology exists, what would have to change over time? For example, the experience of our developers will presumably increase, leading to better productivity and quality. As developers become more experienced and productive, we will naturally have to pay them more, but not proportionally so. Still, there are limits to how productive any one person can be when given a fixed development environment, so the environment too must change.[4]

Over time, we might expect third-party and increasingly open-source software-development tools and libraries to improve, enhancing our development environment and thereby increasing our productivity. While this expectation is reasonable, it will be true for our competitors as well. The question is, "What can we do proactively to improve our productivity relative to the competition over time?"

Implementing a repeatable, scalable software development process has been widely acknowledged to be the single most effective way of simultaneously improving quality while reducing development time and cost. Without such a process, the cost of doing business, let alone the risk of failure, increases nonlinearly with project size. Following a sound development process is essential, yet productivity is unlikely to improve asymptotically by more than a fixed constant multiple. Along with a repeatable process, we also need some form of *positive feedback* in the methodology that will continually amplify development productivity in our environment.

Now consider that there is one essential output of all software development that continues to increase over time: the developed software itself. If it were possible to make use of a significant fraction of this software in future projects, then the prospect for improving productivity could be essentially unbounded. That is, the more software we develop, the more that would be readily available for reuse. The challenge then becomes to find a way to organize the software so that it can and will be reused effectively.

0.2 Application vs. Library Software

Application development is usually single-minded and purposeful. In large organizations, this purposefulness frequently leads both to duplicated code and to sets of interdependent applications. Each piece works, but the overall code base is messy, and each new change or addition becomes increasingly more difficult. This all too frequent "design pattern" has been coined a "Big Ball of Mud."[5]

---

[4] Upon reviewing a near-final draft of this volume, Kevlen Henney remarked, "I have recently been advocating that we ditch the term 'faster' in favour of 'sooner.' It's not the speed that matters, it's the arrival time. These are not the same concept, and the continued focus on speed is a problem rather than a desirable goal. Good design is about taking the better route to arrive sooner; whether you go faster or not is less important."

[5] **foote99**

The resulting code base has no centralized organizational structure. Any software that could in principle be useful across the enterprise either has been designed a bit too subjectively to be generally useful or is too intertwined with the application-specific code to be extricated. Besides, because the code must be responsive to the changing needs of its original master, it would be risky to rely on the stability of such software. Also, because a business typically profits from speed, there is not much of a premium on any of the traditional subdisciplines of programming such as factoring and interface design. Although this ad hoc approach often leads to useful applications in a relatively short time, it also results in a serious maintenance burden. As time goes by, not only is there no improvement in the code base, maintenance costs continue to grow inordinately faster than necessary.

To understand the problem better, we begin by observing that there are two distinct kinds of software — *application* software and *library* software — and therefore two kinds of development. An application is a program (or tightly coupled suite of programs) that satisfies a particular business need. Due to ever-changing requirements, application source code is inherently unstable and may change without notice. All source code explicitly local to an application must, in our view, be limited to use by only that application (see section 2.13).

A library, on the other hand, is not a program, but a repository. In C++, it is a collection of header and object files designed to facilitate the sharing of classes and functions. Generally speaking, libraries are stable and therefore potentially reusable. The degree to which a body of software is particular to a specific application or more generally useful to an entire domain will govern the extent and effectiveness of its reuse within an organization and, perhaps, even beyond.

These contrasting properties of specificity and stability suggest that different development strategies and policies for application and library code should apply. In particular, library developers (few in number) will be obliged to observe a relatively strict discipline to create reusable software, whereas application developers (much more numerous) are allowed more freedom with respect to organizational rules. Given the comparatively large number of application developers who will (ideally) depend on library software, it is critical that library interfaces be especially well thought through, as subsequent changes could wind up being prohibitively expensive.

Classical software design is pure top-down design. At each level of refinement, every subsystem is partitioned independently of its peers. Consideration of implementation at each level of decomposition is deliberately postponed. This process recurses until a codable solution is attained. Adhering to a pure top-down design methodology results in an inverted tree of hierarchical modules (as illustrated in Figure 0-3) having no reconvergence and, therefore, no reuse.

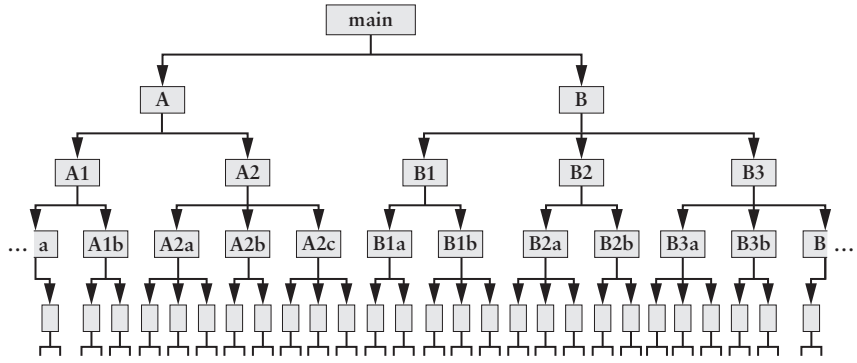


Figure 0-3: Pure top-down design (BAD IDEA)

Although the process of designing any particular application is primarily top-down, experience tells us that, within any given application domain, there are almost always recurring needs for similar functionality. Within the domain of integrated circuit computer-aided design (ICCAD), for example, we might expect there to be many separate needs for classes such as `Transistor`, `Contact`, and `Wire`. There will also be a recurring need for functionality that is common across many application domains. Examples include logging, transport, messaging, marshaling, and, of course, various high-performance data structures and algorithms, such as `std::vector` and `std::sort`, respectively. Failing to recognize common subsystems (and components; see section 0.7) would mean that each must be written anew wherever the recurring need is rediscovered. We assert that it is incumbent on any responsible enterprise to actively address such naturally recurring inefficiencies.

In an ideal application development paradigm, designers would actively seek out commonalities in required functionality across the various subsystems of their programs and, wherever practical, either employ existing solutions or design well-factored components that can serve their recurring needs. Integrating existing solutions into a top-down design makes the design process a hybrid between pure top-down and bottom-up — perhaps the most common architectural approach used in practice today. Even within the scope of a single application, there are typically ample opportunities to apply this kind of factoring for reuse to good effect.

Sadly, reusable software is not normally a byproduct of the development of applications. Because of the focused view and limited time horizon for the great majority of application development efforts, attempts to write software to exploit commonality across applications — absent a separate team of library developers (see section 0.10) — are almost never viable. This observation is not a criticism of application developers but merely reflects a common economic

reality. The success of an application development team is determined by the extent to which it fulfills a business need on time and within budget. Any significant deviation from this goal in the name of reuse is likely to be penalized rather than rewarded. Given the express importance of time to market, it is a rare application developer indeed that makes decoupled, leverageable software available in a form generally consumable by others, while still meeting his or her primary responsibilities.

Library developers, on the other hand, have a quite different mission: Make the overall development process more efficient! The most common goal of library development is to increase the long-term productivity of application developers while reducing maintenance costs (e.g., those resulting from rampantly duplicative software). There are, of course, certain initial costs in setting up a library suitable for public consumption. Moreover, the incremental costs of building reusable components are higher than for similar (nonreusable) ones developed for use in a single application. But, given that the costs of developing library software can be amortized over all the applications that use it, some amount of extra cost can easily be justified.

As Figure 0-4 illustrates, there are several inherent differences between application and library software. Good application software (Figure 0-4a) is generally malleable, whereas library software (Figure 0-4b) needs to be stable (see section 0.5). Because the scope of changes to an individual application is bounded, the design of the application's pieces is often justifiably more tightly collaborative (see section 0.3) than would be appropriate in a library used across arbitrarily many applications.[6]

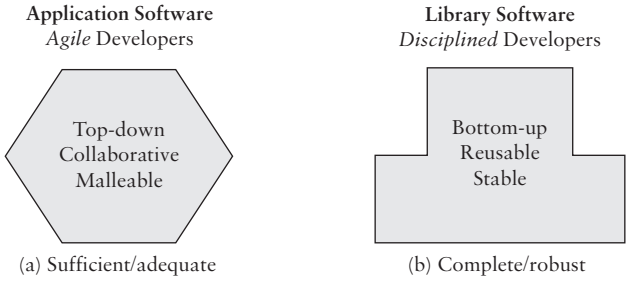


Figure 0-4: Library versus application software development

[6] See also **sutter05**, item 8, pp. 16–17.

The requirements for library software are, in many ways, just the union of those of the applications that depend on it (see Volume II, section 5.7). For example, the lifetime of a separately releasable library is the union of the lifetimes of the applications that use it. Hence, libraries tend to live longer than individual applications. If an application is to be released on a particular platform, then so must any library that supports it. Therefore, libraries must be more portable than applications. Consequently, there will often be no single application team that can support a given library throughout its productive lifetime on all the platforms for which it might be needed. This observation further suggests the need for a separate, dedicated team to support shared resources (see section 0.10).

Library code must be more reliable than typical application code, and our methodology amplifies this dichotomy. For example, compared with typical application code, library code usually has more detailed descriptions of its programmatic interfaces, called *contracts* (see Volume II, section 5.2). Detailed function-level documentation (see Volume II, section 6.17) permits more accurate and thorough testing, which is how we achieve reliability (see Volume II, section 6.8, and Volume III in its entirety).

Also, library software is more stable, i.e., its essential behavior does not change (see section 0.5). More stability reduces the likelihood that bugs will be introduced or that test cases will need to be reworked due to changes in behavior; hence, stability improves reliability (see Volume II, section 5.6). Having a comparatively large number of eclectic clients will provide a richer variety of use cases that, over time, tends to prove the library software more thoroughly. Given that the cost of debugging code that you did not write (or write recently) is significantly higher than for code you are working on today, there is a strong incentive for library developers to "get it right the first time" (see Volume III, section 7.5).

When writing library software, we strive to absorb the complexity internally so as to minimize it outwardly. That is, library developers aggressively trade off ease of implementation (by them) for ease of use (by their clients). Small pockets of very complex code are far better than distributed, somewhat complicated code. For example, we assert that it is almost always better to provide two member functions than to provide a single template member function if only two parameter types (e.g., consider `const char *` and `std::string`) make sense (see Volume II, section 4.5).

More controversially, it is often better to have two copies of a `struct` — e.g., one nested/private in the `.h` file (accessible to `inline` methods and friends) and the other at file scope in the `.cpp` file (accessible to file-scope `static` functions) — and make sure to keep them in sync locally than to pollute the global space with an implementation detail. In general, library developers should plan to spend significant extra effort to save clients even a slight

inconvenience. The more general the reusable component, the more developer effort can be justified (see Volume II, Chapter 6).

As with any business, the focus of ongoing library development must be to demonstrate significant value at reasonable incremental cost. Hence, any library software that is written must be *relevant* to those who would use it. Library developers need look no further than to existing applications for guidance. By working closely with application developers to solve their recurring needs, library developers stay focused on what is important: making application developers more productive!

At the same time, library developers have the important responsibility to *judiciously* and *diplomatically* resist inappropriate specific requests that would undermine the success of the development community as a whole. For example, refusing to provide a `name` attribute for a `Calendar` type might seem capricious to an application developer scrambling to make a deadline. Yet the experienced library developer knows (see Volume II, section 4.3) that providing a `name` field will destroy any useful notion of *value* for that object and render it unsuitable as the *vocabulary type* (see Volume II, section 4.4) that it was designed to be.

Even valid requests are sometimes phrased in terms of a proposed solution that is suboptimal. Instead of blindly satisfying requests exactly as they are specified (see section 3.12.1), it is the responsibility of library developers to understand the underlying needs, along with *all* of the software issues, and provide viable solutions that meet those needs (if not the wants) of all clients — both individually and collectively. Hence, the design of the client-facing interface is critically important (see Volume II, Chapter 5).

The design paradigms for application and library development are also quite different. Whereas application design is primarily top-down, libraries consisting of generally useful software are developed bottom-up. More sophisticated solutions are composed out of simpler ones. The desirable tree-like — or more accurately DAG-like — dependencies among libraries and even among the components within those libraries are not a coincidence. They are instead the result of thoughtful factoring for the express purpose of achieving fine-grained *hierarchical reuse* (see section 0.4) *across* applications.

The absolute need for clear, concise, and complete documentation also differentiates library code from its application counterpart. Since library code exists independently of any one context, far more thorough and detailed overview documentation (see Volume II, section 6.15), along with thoughtful usage examples (see Volume II, section 6.16), will be needed for clients

to readily understand and use it. And, unlike applications, the performance requirements for reusable library software are not bound to any one usage and often must exceed what is provably needed today (see section 0.11).

Deployment is yet another aspect in which application and library software diverge. An application is typically released as a single executable, whereas libraries are deployed as collections of header files (see section 1.1.3) and corresponding library archives (see section 1.2.4) or, perhaps, shared libraries. While incorporating directory structure (e.g., via relative pathnames) in `#include` directives (e.g., `#include <basic/date.h>`) might be acceptable for application software, libraries should abstain from doing so, or flexibility during deployment will be compromised (see section 2.15).

Finally, application developers are invariably attempting to solve difficult, real-world problems. As Figure 0-5 illustrates, these problems are often irregular and particularly onerous to describe in all their detail. When developing a large application top-down, there is often uncertainty regarding aspects of the overall solution — perhaps some important requirements are missing or yet to be determined. Moreover, these intricate problems — and therefore the underlying application software itself — will typically grow and change over time.

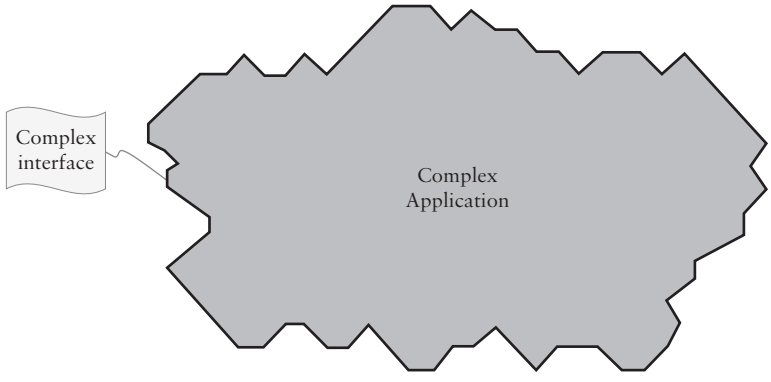


Figure 0-5: Many applications aren't regular.

Still, within such a complex application there are typically many regular, well-defined, stable subproblems whose factored solutions will find their way into the application. In such situations, we might be able to jump-start the development process by sponsoring the creation of some generally useful pieces. Working together, application and library developers might be able to identify common subsystems that help shape the requirements and/or design of the

application, as illustrated in Figure 0-6. Interaction between application and library developers helps to galvanize crisply specified contracts that are relevant and generally usable, and perhaps might even be reusable (see Volume II, section 5.7).[7,8]

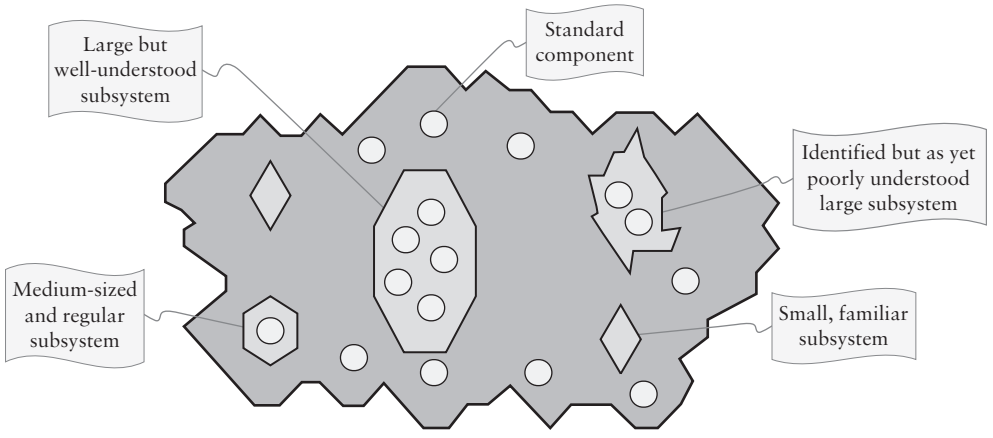


Figure 0-6: Many components and subsystems *are* regular.

By routinely solving slightly more general, more easily described subproblems that address most — but not necessarily all — aspects of a specific need, we will systematically achieve

---

[7] See also **stroustrup00**, section 23.4.1, pp. 698–700, which we summarize here.

In his book, Stroustrup makes a number of sound recommendations and observations.

- Design using existing parts.
- Hold off on creating new, eccentric, nonreusable custom parts.
- Try to make any new parts you create useful in the future.
- Only in the end create project-specific parts.

Stroustrup goes on to say that this approach can work, but it is not automatic.

- Often we cannot even identify distinct components in the final design.
- "...corporate culture often discourages people from using the model outlined here."
- "...this model of development really works well only when you consider the longer term."
- A universal/international standard for all components is not reasonable.
- A hierarchy of standards — country, industry, company, department, product — is the best we can realistically hope to achieve.
- Iteration is essential to gain experience.

[8] As noted in the preface, many of the references in this chapter in particular, such as the previous one, are to books that have had subsequent editions that "supersede" their earlier ones, but those later editions fail to carry forward the sage design advice due to length constraints. We therefore continue to reference these early (perhaps out-of-print) editions and, of course, reproduce the relevant content here for the convenience of the reader.

less *collaborative* (mutually *aware* — see section 0.3), more *stable* software, and hence more *reusable* software, even as our applications continue to evolve. Like thermal separators between concrete blocks, the gaps in functionality between assembled *reusable* components (filled in by glue logic) are what accommodate the less regular, more fickle, and ever-changing policies that large applications typically comprise (see Volume II, section 5.9).

To achieve maximal reuse, an organization must make an ongoing effort to factor out of application-specific repositories (illustrated in Figure 0-7a) as much independently comprehensible code as it can (e.g., as illustrated in Figure 0-7b). The more widely applicable that code might be, the lower in the firm-wide repository of reusable software it belongs. Only the most widely useful software would reside near the root.

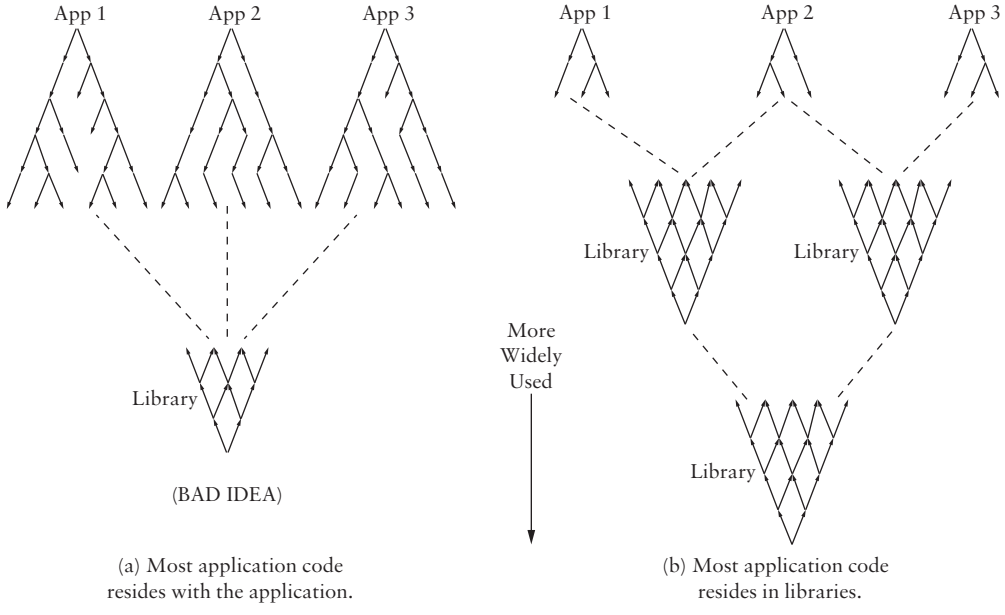


Figure 0-7: Relative application versus library sizes

Given the appropriate mandate, detail-oriented *disciplined* library developers, working closely with fast-moving *agile* application developers, have a far better chance at rendering the kind of noncollaborative, high-quality, stable pieces of software that lead to effective reuse. It is by proactively extracting an appropriate subset of the software we develop, aggressively refactoring it,

and placing it into stable, accessible, and therefore potentially reusable libraries, that we might achieve substantial productivity gains in the long term.

0.3 Collaborative vs. Reusable Software

The criteria for the modularization of classically reusable software emphasize isolating manageable amounts of complexity behind interfaces that minimize collaborative idiosyncrasies. The industry recognizes that continuous refactoring — i.e., extracting, refining, and *demoting* (see section 3.5.3) widely usable functionality as it is discovered — is the only viable approach for harvesting anything more than the most basic reusable software. Even when reuse per se is not the goal, careful factoring can provide significant flexibility, leading to greater maintainability as the development effort progresses, especially as requirements change. When designing top-down, however, many developers routinely construct obscure devices that exactly meet their current needs but that are highly specific, inflexible, and clearly not useful in other contexts. The metaphors we use to characterize such poorly or inadequately factored subsystems are, respectively, the *cracked plate* and the *toaster toothbrush*.

Factoring is the art of subdividing a larger problem, illustrated schematically in Figure 0-8, into appropriate smaller ones whose independent solutions yield useful components that can be explained, implemented, tested, and recombined to solve the original problem. In a well-factored solution, the complexity of the interface for each subsolution will be kept small relative to the complexity of the corresponding implementation that does the real work. Analogous to a sphere, the ratio of complexity across its interface (or *surface*) to that of its implementation (or *volume*) is minimized.

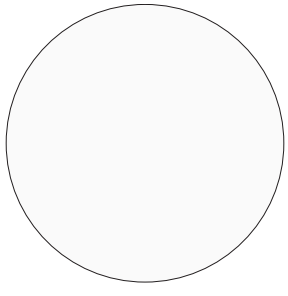


Figure 0-8: A large but well-defined software problem

In the worst case, the complexity of interactions across module boundaries will be no simpler than those within. To use such a module, clients will need to know essentially all details of the underlying implementation. Any change to that implementation — no matter how small — runs the substantial risk of adversely affecting the desired behavior of any of the other modules with which it interacts. Such inadequacies in logical factoring result in physical modules that can be extraordinarily difficult to maintain.

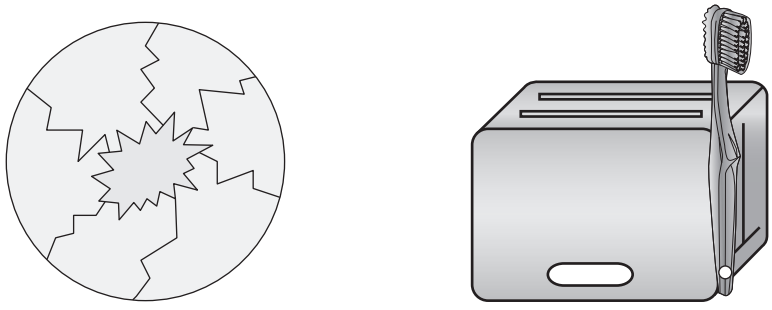
For example, imagine that we have a subsystem consisting of a suite of modules that all communicate through a common message buffer implemented as an `std::vector<char>`. Initially, the byte offsets are blocked off to correspond to a well-known structure: The first four bytes indicate the source of the transmission, the second four bytes indicate the destination, the third four bytes indicate the transmission type, and so on.

Over time (and always for valid business reasons), this structure is slowly but surely corrupted. One module adds the rule that if the most significant bit of the transmission type is set, use the "alternate enumeration" for source and destination. Some other modules, having no use for some of the fields, redefine their meaning under certain stateful conditions known only to them. Still others, not wanting to "corrupt" the message format, pre-allocate additional capacity and store "private" information beyond the logical end of the vector.[9] It remains, however, a common practice to steal the first two (or perhaps three) least significant bits (depending on the target architectures) of a pointer, based on the very practical (but not guaranteed) assumption of *natural alignment* (see Volume II, section 6.7).

What was once a relatively clean interface has become overwhelmingly complicated. The unconstrained, and now tight, logical coupling among these modules makes even minor maintenance tasks both expensive and risky. Like the pieces of a cracked plate, these modules are destined to serve exactly one useful purpose — i.e., implementing the current version of exactly one application. The resulting "partition" is illustrated schematically in Figure 0-9a.

---

[9] A previous boss of mine at Bear Stearns, Buzz Moschetti, sardonically referred to this insidious form of extreme brittleness (c. 1998) as a *zvector*.



(a) Cracked plate (b) Toaster toothbrush

Figure 0-9: A brittle solution

A common result of inadequate factoring is a function, object, or component that is both unfamiliar and difficult to describe. Being difficult to describe is a telling symptom of a large *surface area* (i.e., complicated interface) and a strong indication that the original problem was improperly or insufficiently factored. Disproportionate complexity in the interface results when a module is pathologically specific to the current need.

Let us now imagine that your mom is the client and you have been given a set of requirements for the morning: Brush your teeth and make toast for the family. Being the industrious type, you quickly invent a solution to both requirements as a single seamless device cast from one mold, the "toaster toothbrush" (Figure 0-9b). Although simultaneously addressing precisely the two requirements delineated by customer Mom, a `ToasterToothbrush` — much like each of the pieces of a cracked plate — is unique to the circumstances that precipitated it and is therefore entirely ineffective in the presence of change. A minor modification to any aspect of the original problem specification (e.g., Figure 0-8) could require us to revisit implementation details in every corner of the existing solution (e.g., Figure 0-9a). As it is typical for application requirements to change over time, such brittleness in the pieces of the application presents an enormous liability and a serious risk to a timely success: If your mom changes her mind — even just a little — you're toast!

Even when pathologically brittle designs are avoided, it is not unusual for application developers to create software that, while nominally independent, was clearly conceived with peer components in mind. Let's skip ahead and consider first the collaborative decomposition of the toaster-toothbrush application illustrated in Figure 0-10b. The familiar toaster object has been thoughtfully augmented with a hook that provides a handy location at which to attach the requisite toothbrush — provided, that is, the toothbrush in question has been properly customized with an appropriate hole bored through its handle. Apart from this rather specific circumstance, it is unlikely (all other things being equal) that one would opt for a toaster with an unsightly hook protruding from it to reside on their kitchen counter.

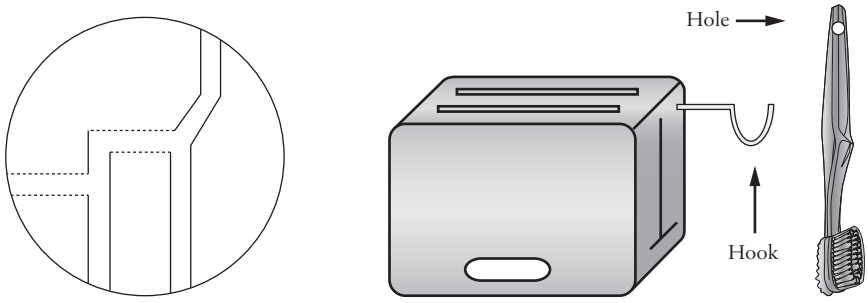


Figure 0-10: Collaborative solutions

Circling back to our plate analogy, it is also possible to design a software solution that has a well-defined interface and a low surface-to-volume ratio yet is nonetheless composed of highly collaborative pieces, as illustrated in Figure 0-10a. This circle, like the toaster toothbrush, is an example of a *collaborative* design — one in which, even though there might be no explicit *physical* interdependency among the submodules, the *logical* design of each piece was highly influenced by its surroundings. As you can see, fully describing any one of these pieces alone, without reference to the others, would take some doing. Like a toaster with a hook, or a toothbrush with a hole, such pieces are substantially less likely to find their way into wider use in new contexts than a less collaborative, more easily described, more *reusable* set of components.

Still, collaborative designs can be acceptable within the context of a single application, particularly if we are able to reuse the collaborative pieces across multiple versions. Compare the cracked plate of Figure 0-9a to the collaborative one of Figure 0-10a. A piece of a cracked plate is unlikely to be useful except to fit back together in the specific version of the software for which it was designed. As Figure 0-11a illustrates, brittle solutions such as this are usually limited to a single version of a single application; all of its parts must be reworked in unison with each new release. Simply put, this less-disciplined approach to software development is not effective on a large scale, which perhaps explains why so many larger projects fail (or fail to meet expectations).

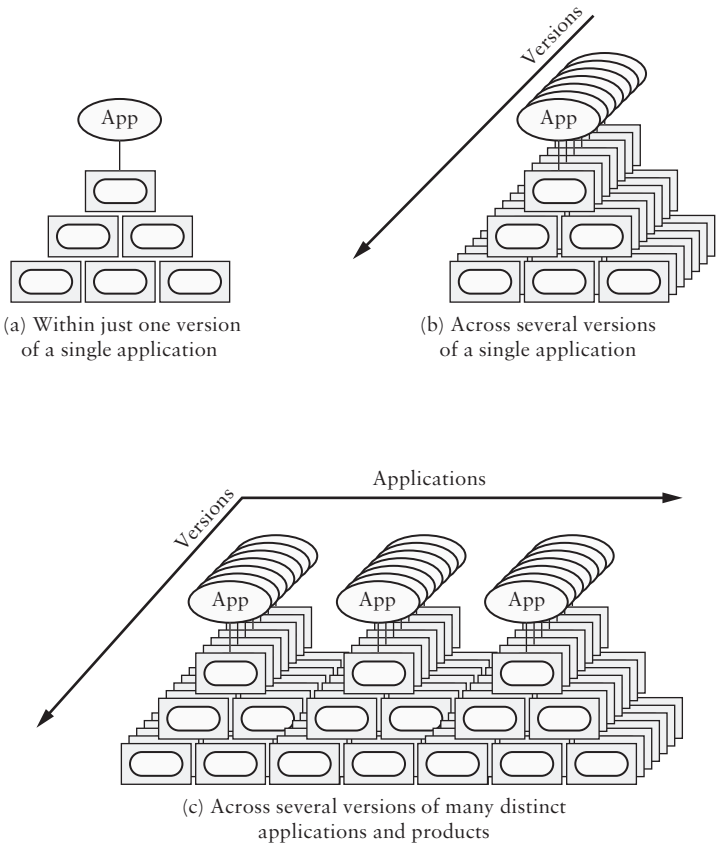


Figure 0-11: Degrees of exploiting solutions to recurring problems

However, through careful factoring, we can reuse collaborative pieces *across* versions (Figure 0-11b). These pieces can reside in application-specific libraries outside of the application source code, where they can be shared *without modification* across multiple active versions of the application. By increasing the proportion of subfunctionality that remains stable across versions, we reduce not only the overall costs associated with ongoing enhancements, but also the time needed between releases.

While collaborative solutions might be reusable across versions of a single application, to achieve reuse across applications (Figure 0-11c), we must go further. Historically, *reusable* software addresses a small, well-defined problem that recurs in practice. Familiar examples

include `std::vector` and `std::string` from the C++ Standard Library. In our geometric analogy, reusable software components take the form of familiar shapes, as illustrated in Figure 0-12a. Notice that each of these shapes is easy to describe in words: a *square* of side *S*, a *rectangle* of length *L* and width *W*, a (right) *triangle* of base *B* and height *H*, and a *semicircle* of radius *R*.

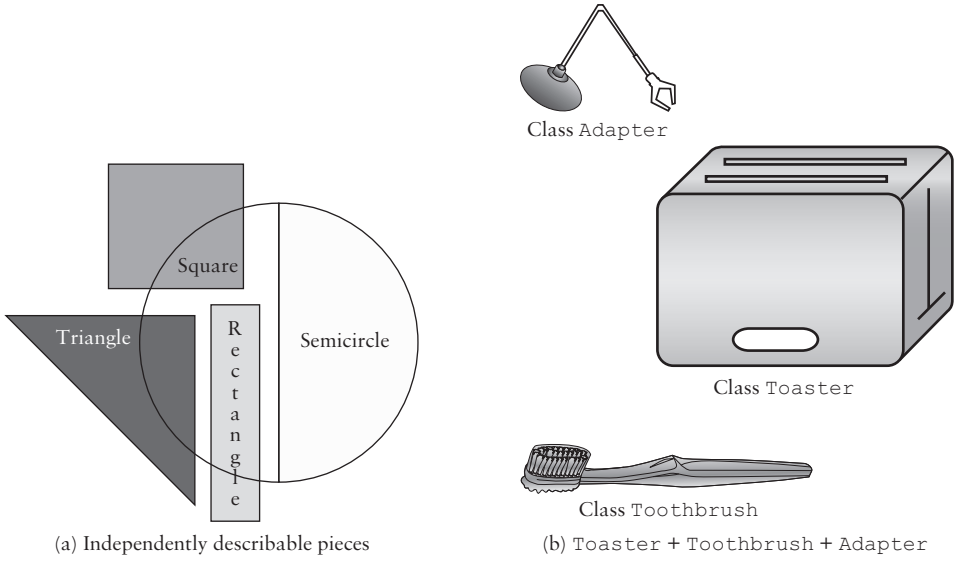


Figure 0-12: Classically reusable software

Reusable software must be easy to compose *without modification* (see section 0.5) in various ways to solve new, unforeseen problems. Having a trusted suite of prefabricated components can be a tremendous productivity aid. Assuming we had an "off-the-shelf" toothbrush (sans hole), an ordinary toaster (sans hook), and the interesting (but definitely noncollaborative) adapter of Figure 0-12b, we could readily assemble a solution very much resembling the fully custom, monolithic device of Figure 0-9b, but much more quickly *and reliably* than building it from scratch. As illustrated in Figure 0-13, such noncollaborative subsolutions will typically require some filler (Figure 0-13a); however, a bit of glue (logic) is often all that is needed (Figure 0-13b).

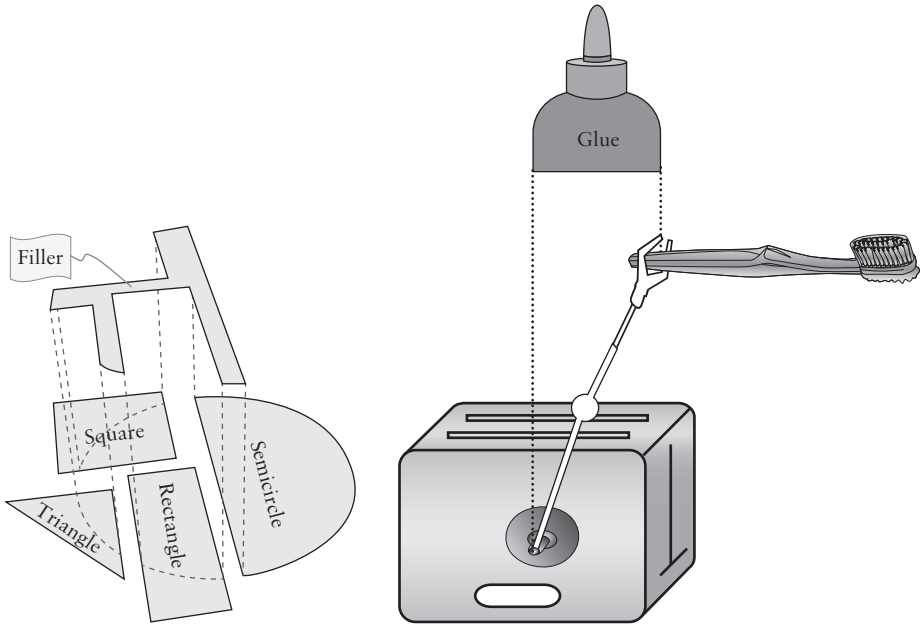


Figure 0-13: Solving the original problem with classically reusable pieces

It is when we consider the value of a finely graduated hierarchy of truly *reusable* solutions over many versions of many distinct applications and products (Figure 0-11c) that we can begin to appreciate the vast potential benefits of component-based software in general (see section 0.7) — and *Software Capital* in particular (see section 0.9). As we will soon illustrate, it is this two-dimensional scenario that compels us to allocate the resources necessary to construct and populate a viable firm-wide repository of ultra-high-quality, *hierarchically* reusable software.

0.4 Hierarchically Reusable Software

For quite some time now, people have known that large monolithic blocks of code (illustrated in Figure 0-14) are suboptimal. As far back as 1972 — before the promise of reuse and some quarter century before physical software design concepts would come to be addressed as such — luminaries such as Parnas[10] observed that carefully separating even just the logical content of a program into its more tightly related subsystems and permitting only simple (e.g., scalar) types to pass through their interfaces alone tended to reduce overall complexity.

---

[10] D. L. Parnas is widely regarded as the father of *modular programming* (a.k.a. encapsulation and information hiding); see **parnas72**.

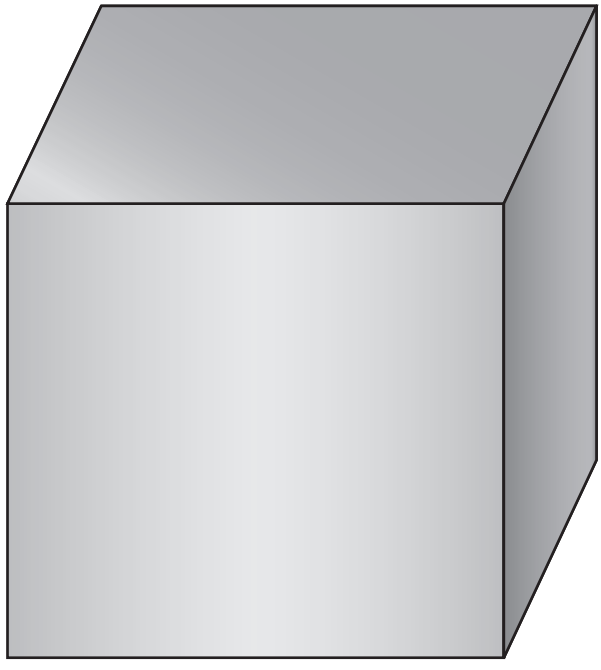


Figure 0-14: Monolithic block of software (BAD IDEA)

Some four years previous (1968), Dijkstra — known for his many important contributions to the field of mathematical computer science, such as the shortest-path algorithm that bears his name[11] — had already published the results of a body of work[12] that demonstrated the value of a hierarchical (acyclic) physical software structure for enabling the comprehension and thorough testing of complex systems. Parnas, in his seminal paper,[13] acknowledged that an (acyclic) physical hierarchy (illustrated in Figure 0-15a) along with his proposed "clean" logical decomposition (illustrated in Figure 0-15b) "are two desirable, but *independent* properties of a system structure."[14]

---

[11] **dijkstra59**

[12] **dijkstra68**

[13] **parnas72**

[14] Observing the increase in flow-of-control switches between modules in his own (then new) decomposition style, Parnas went on to correctly forecast the need for `inline` function substitution (see section 1.3.12) for modular programs to remain competitive at run time with their procedural counterparts.

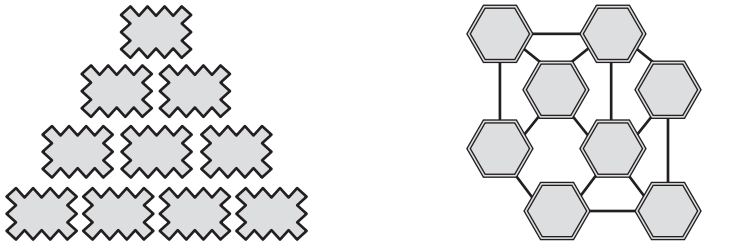


Figure 0-15: Two desirable properties of a system structure

A few years later (1978), Myers would argue,

> ... a more powerful justification for partitioning a program is that it creates a number of well-defined documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program.[15]

Although Myers happened to be referring to the maintainability of just a single program, these arguments naturally apply more generally across any physical boundaries — e.g., libraries.

Classical software design often results in a coarsely layered architecture, as illustrated in Figure 0-16. An insufficiently factored architecture such as a naive implementation of the seven layers of the OSI network architecture model[16] — or even the various layers of a compiler — is not necessarily a problem so long as the layers themselves readily admit further vertical decomposition.[17]

---

[15] **myers78**, Chapter 3, "Partitioning," pp. 21–22, specifically the last paragraph of p. 21

[16] **deitel90**, section 16.2, pp. 496–515

[17] Note that the OSI model suffers in that some "implementations" have distributed the tasks of the layers in a different way or added sublayers. An example is Ethernet II (layers 1+2) versus IEEE 802.3+802.2 where layer 2 was basically split up into 2a and 2b, adding an extra header. (This data courtesy of JC van Winkel.)

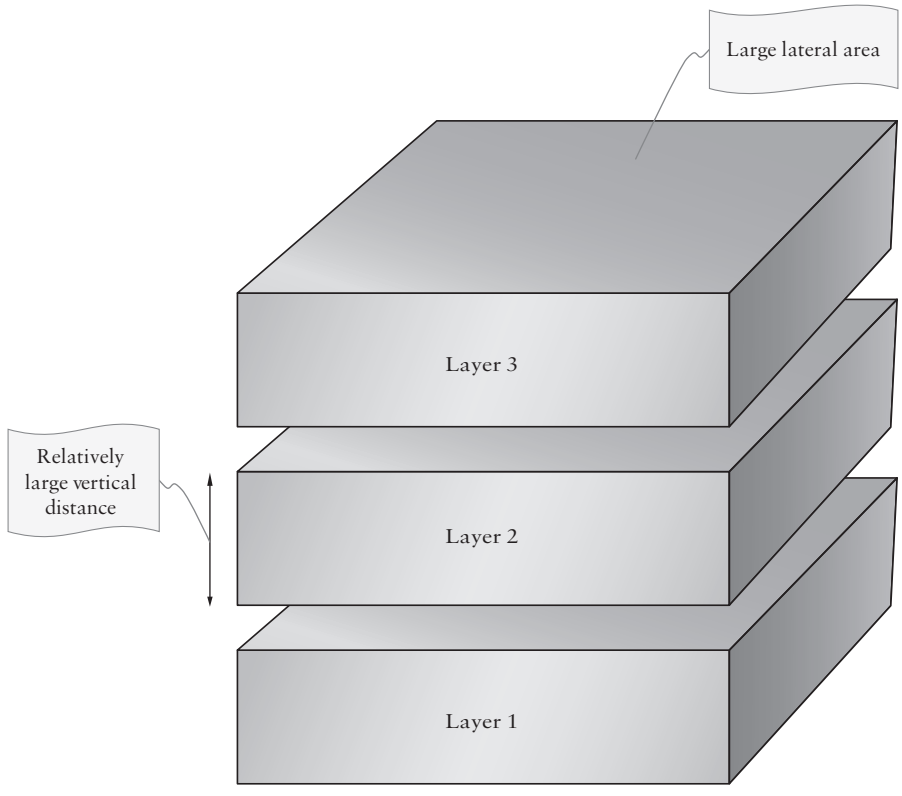


Figure 0-16: Coarse-layered software structure

A more finely graduated approach to vertical factoring is illustrated in Figure 0-17. By *finely graduated*, we mean that the amount of functionality added from one accessible level of abstraction to the next is relatively small. For example, class `Line` uses class `Point` in its interface (and therefore depends on it), but the definition of `Line` resides in a separate physical unit at a (slightly) higher level. Providing "short" logical steps from one physical level to the next dramatically improves the thoroughness with which we can understand and test our systems.

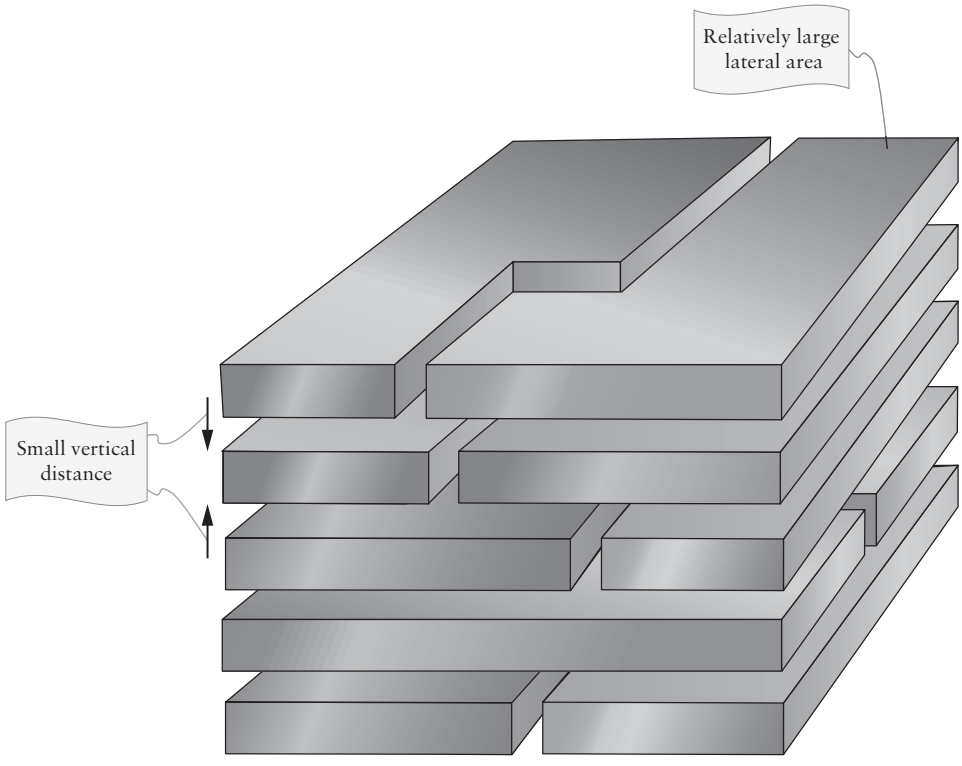


Figure 0-17: Finely graduated (but nongranular) software

Finely graduated vertical factoring alone, however, is not sufficient for efficient component-level testing (see Volume III, section 7.5) nor, due to excessive physical dependency, is it suitable for independent reuse (see section 1.8). We must also make sure to separate peer logical content — e.g., the respective definitions of `Circle`, `Triangle`, and `Rectangle` — into physically separate units (see section 3.2).

Providing broad, albeit shallow, logical content — such as a wrapper layer — will likely touch many aspects of the lower layers. If this layer resides in just one or a few large monolithic physical units, then functionality not directly needed for the current purpose will nonetheless draw in other unneeded functionality (see section 1.2.2). The result is that a disproportionately large amount of code must be compiled and linked in order to test or reuse any particular unit of logical content. The likely physical consequences are increased compilation time, link time, and executable size (see section 3.6), resulting in larger (and often slower) programs. To mitigate these problems, we must be careful to partition our logical designs laterally as well as vertically.

Classically reusable software is granular; its laterally modular structure is illustrated in Figure 0-18. By *granular*, we mean that the breadth of logical functionality implemented in any one atomic physical unit is small. The codified solution to a well-defined, recurring problem is packaged in such a way as to be readily accessible to clients yet physically independent of anything not required. We must be careful, however: If this packaging goes too far and ends up nesting (and therefore rendering inaccessible) the subfunctionality used to implement the granular functionality, the finely graduated vertical aspect is lost and with it goes *hierarchical* reuse.

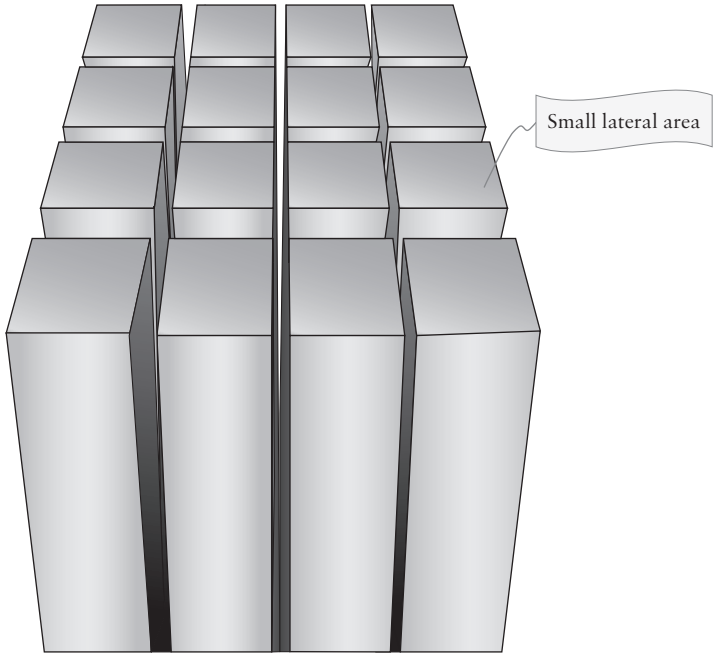


Figure 0-18: Granular (modular) but nongraduated (not finely layered) software

In practice, classical library reuse has tended to be bimodal: Reusable solutions are either tiny (e.g., algorithms and data structures) and devoid of policy (i.e., a specific context governing when, where, and how to use this functionality) or large and influential (e.g., databases, messaging middleware, logging systems) that more forcefully dictate their intended use.[18]

[18] Frameworks provide an important, but substantially different, *top-down* form of very coarse-grained, structured reuse.

Little in the way of intermediate subsolutions, however, have been broken out, documented, and made generally available.

Using the *component* (see section 0.7) as the atomic unit of logical design and physical packaging, the *finely graduated*, *granular* software structure illustrated in Figure 0-19 incorporates the benefits of fine-grained physical modularity (both horizontally and vertically) with a clean and well-documented (see Volume II, sections 6.15–6.17) logical decomposition. Like the thin layers of Figure 0-17, this organization facilitates thorough testing. And, like the independent solutions of Figure 0-18, these individual components depend on only what is needed — nothing more.

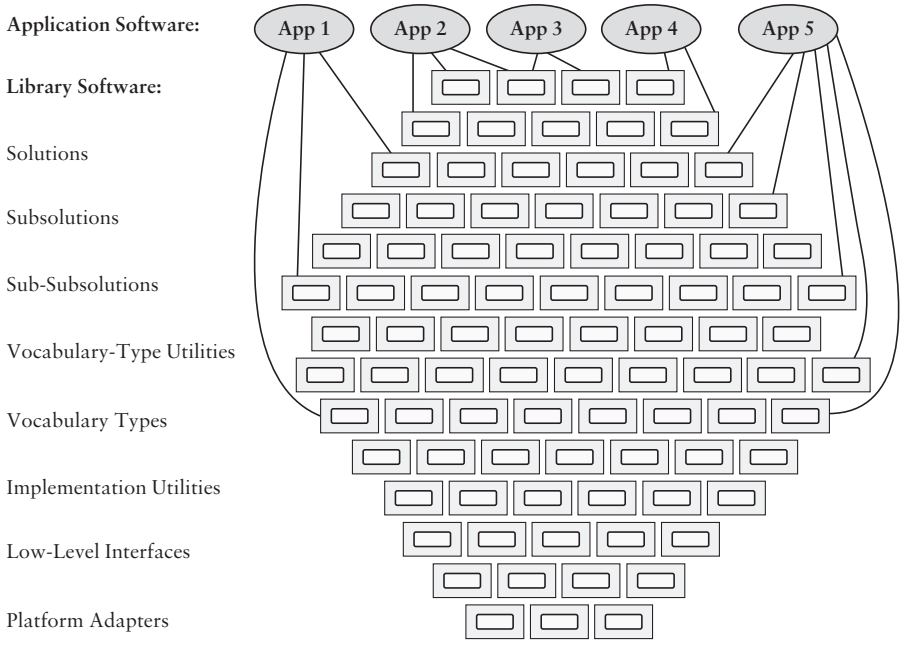


Figure 0-19: Finely graduated, granular software

If designed properly, a finely graduated, granular solution will be built from a suite of well-factored subsolutions, which in turn draws from sub-subsolutions and so on.[19] We can easily reveal, like peeling an onion, the inner levels of abstraction that, without modification (see section 0.5), provide well-factored subproblem solutions common to both the original need and often emerging new ones.

[19] See also **stroustrup00**, section 24.3.1, pp. 733–734.

For example, common types — especially those that flow through function boundaries, such as `Date` and `Allocator`, which we call *vocabulary types* (see Volume II, section 4.4) — are obvious candidates for reuse. These basic types might at first appear to be at one end of the reusable spectrum. Yet, virtually all concrete types are naturally layered on significantly lower-level infrastructural functionality.

Part of what makes this approach to reuse so compelling is that, unlike the software structure of classical reuse (e.g., Figure 0-12), the client is not limited to a single layer of abstraction. Recall our classically decomposed toaster, toothbrush, and adapter from Figure 0-12b. We were able to easily build the toaster toothbrush out of these parts (Figure 0-13b). However, by extending the process of factoring software across multiple finely graduated layers (as illustrated in Figure 0-20a), we are able to make available a much richer set of interoperable functionality (Figure 0-20b). This low-level functionality can be readily recombined to form new higher-level *reusable* subsolutions at substantially reduced costs — i.e., both the cost of development *and* the cost of ownership. These small, fully factored subimplementations (see Volume II, section 6.4) also lead to improved quality by enabling thorough testing (see Volume III, section 7.3).

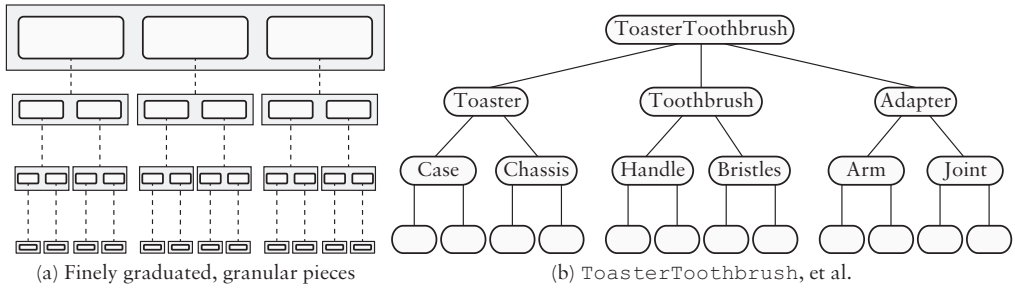


Figure 0-20: Hierarchically reusable software

Consider the fully factored, fine-grained hierarchy of Figure 0-21, initially consisting of a `Toaster`, `Adapter`, and `Toothbrush`. Now suppose that we are asked to create a `ToasterFlashlight` and a `ToasterScrubbrush`. Given the classical reusable implementation of Figure 0-12b, we can easily construct a toaster flashlight by building a standard `Flashlight` type and then recombining the other two standard pieces — i.e., the toaster and the adapter — with the new flashlight to form a `ToasterFlashlight`. Proper factoring, however, will lead to additional components used to implement the flashlight — i.e., those defining `Lens` and `Stock` (Figure 0-21a).[20]

[20] A nice, real-world example, suggested by JC van Winkel, is that of *Phonebloks* in which a phone is built from bricks that are replaceable — e.g., don't need a camera, put in bigger battery. See www.phonebloks.com and www.youtube.com/watch?v=oDAw7vW7H0c.

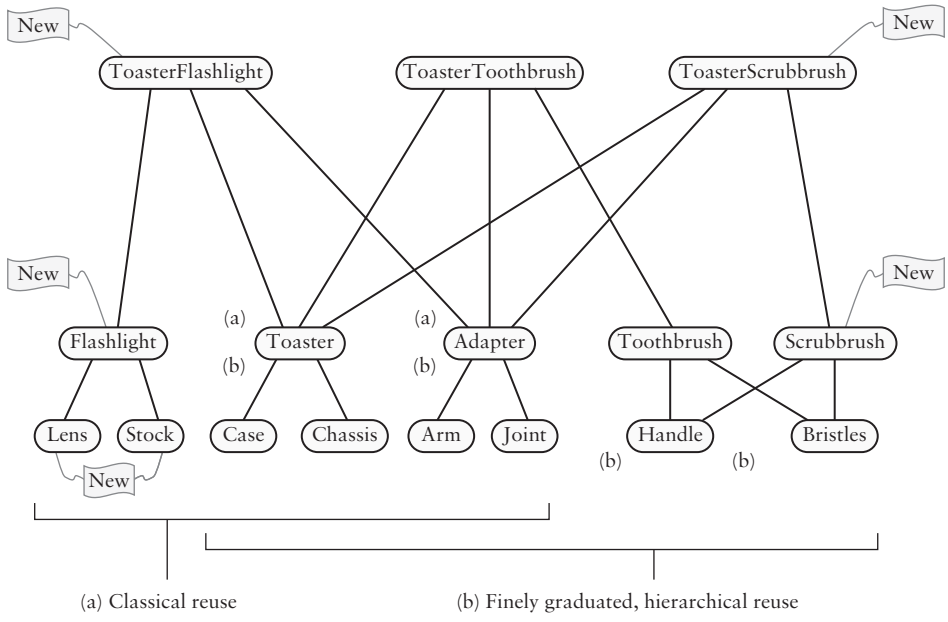


Figure 0-21: Exploiting finely factored library software

If, however, all that is needed is a toaster scrubbrush, we might be able to apply a bit more "glue" to create a new scrubbrush out of precisely the same primitive materials (i.e., `Handle` and `Bristles`) used to assemble the toothbrush! Leaving `Toothbrush` unaffected, we can then combine the new `Scrubbrush` type with the toaster via the adapter and *voilà* — a `ToasterScrubbrush` (Figure 0-21b).

Fine-grained physical modularity is essential to effective reuse. It is only by imposing fine graduations on granular solutions that we enable independent reuse at every level of abstraction as opposed to just data structures and subsystems. Moreover, it is this highly factored, hierarchically modular physical infrastructure that enables direct access (in place) to what would otherwise be inaccessible subsolutions and, therefore, have to be duplicated.

The property that distinguishes our hierarchically reusable software from conventional software is that we practically never need to *change* what we have in order to respond quickly to new situations. The more mature our repository of hierarchically reusable software solutions, the more quickly we can execute because more and more of the work has already been done for us (see section 0.9). In section 0.6, we will take a closer look at physical design aspects of our fine-grained software libraries, but first, in the next section, we explore the importance of stability to effective reuse.

0.5 Malleable vs. Stable Software

Good software generally falls into one of two broad categories: malleable or stable. *Malleable* software, by definition, is software whose behavior is easily and safely changed to reflect changing requirements. *Stable* software, on the other hand, has well-defined behavior that is deliberately *not* altered incompatibly in response to changing client needs.

The behavior of stable software can, by coincidence, also be easy to change; sound software engineering practices that facilitate malleability, such as well-documented symbolic constants (as opposed to magic numbers), of course apply to stable code as well. The important difference is that the behavior of malleable software is intended to change, whereas the behavior of stable software is not. Both kinds of software have their place, but software that is neither malleable nor stable is undesirable. Most of all, it is important never to confuse *malleable* with *reusable*. Only stable software can be reusable.

Unlike library software, the behavior of application code typically needs to be able to change quickly to respond to rapidly changing business needs. Hence, well-designed application code is malleable. Just because something is likely to change in the future does not mean that we should not expend significant up-front thought before we begin to code. Unfortunately, some well-intentioned people have treated the object-oriented paradigm as if it were a license to write code without thinking: "Let's implement some objects, and we can refine the details later." This approach might be a useful exercise in an introductory programming course, but it is far from anything we would call *engineering*.

At the far end of the "rapid" development spectrum is a general methodology known as *agile software development*[21] of which *extreme programming* (XP)[22] is an early special case. The original, stated goal of this approach[23] was to address each emerging need as quickly as possible and in whatever way maximizes current business objectives. All forms of bureaucracy are to be shunned. The extreme/agile programmer epitomizes the best of application programming by delivering the highest-quality product possible with the allotted resources on schedule. The concession is that the long term will have to be addressed later, and the software that agile programmers do write themselves will most likely soon have to change incompatibly as a result.

---

[21] **cockburn02**

[22] **beck00**

[23] **alliance01**

Most agile programmers will freely admit that they would be happy to use whatever software is available that allows them to meet their objectives. Given the opportunity to make use of a stable set of hierarchically reusable libraries, truly agile programmers most certainly would. Even the most agile of programmers cannot compete with other similarly agile programmers who, in addition to their agility, also have access to a mature library of relevant, prefabricated, stable solutions.

Without detracting from this widely practiced approach to software development, we do want to remind our readers that there is quite a bit more to this story.[24] Although agile techniques can be useful in developing modest-sized applications quickly in the short to medium term, they are generally not suitable for developing software on a large scale — especially (stable) reusable software.

Imagine, once again, that we find ourselves in need of a toaster, so we march down to our local hardware/home-improvement store and buy one. Now imagine to our surprise that one day while attempting to make toast we discover that our "dependable" toaster has morphed into, say, a toaster oven or, worse, a waffle iron. No matter how much better the mutated mechanism becomes, unless it continues to behave as originally advertised for all existing clients, whatever plans or assumptions we have made based on having a toaster are immediately called into question. It is hard to imagine this problem in real life, but sadly, this sort of thing can and often does occur in the realm of software.

Classical design techniques lead directly to software that must be modified to maintain it. By "maintain" here we sarcastically mean making a piece of code do what we want it to do *today* as opposed to what we wanted it to do *yesterday*. For example, consider the following function `f` designed to solve some common real-world problem:

```
f(a, b, c, x, y, z)
```

What led us to believe that six parameters were enough? What happens if, over time, we discover (or, more likely, are asked to consider) additional factors `d` and perhaps even `e`. Do we go back and change function `f` to require the additional arguments? Should they be optional?

In virtually all cases, the answer to these questions is no. If `f` is supposed to implement a documented behavior and is currently in use by several clients, we simply cannot go back and change `f` to do something substantially different from what it did before, or nothing that

[24] **boehm04**

depended on `f` would work anymore. By some metrics,[25] the very measure of *stability* for a given entity derives from the number of other entities that depend on it. Ideally, any change we consider would leave all of our clients' original assumptions intact (see Volume II, section 5.5).

The problem here results from a fundamentally over-optimistic approach to design: We allowed ourselves to think that there are ever enough parameters to solve any interesting real-world problem; there usually are not — not for long. The root cause of the instability can typically be traced to inadequate factoring. A more stable design would aggressively factor the implementation into a suite of simpler functions, each of which has a reasonable chance of being complete in and of themselves:

```
fa(a)   fb(b)   fc(c)   fx(x)   fy(y)   fz(z)
```

As Figure 0-22 illustrates, given a more finely factored, granular design (Figure 0-22a), if we need to add new functions `fd(d)` and `fe(e)`, we might be able to leave the other functions currently in use untouched (Figure 0-22b).[26] This way, no pre-existing software applications are affected.

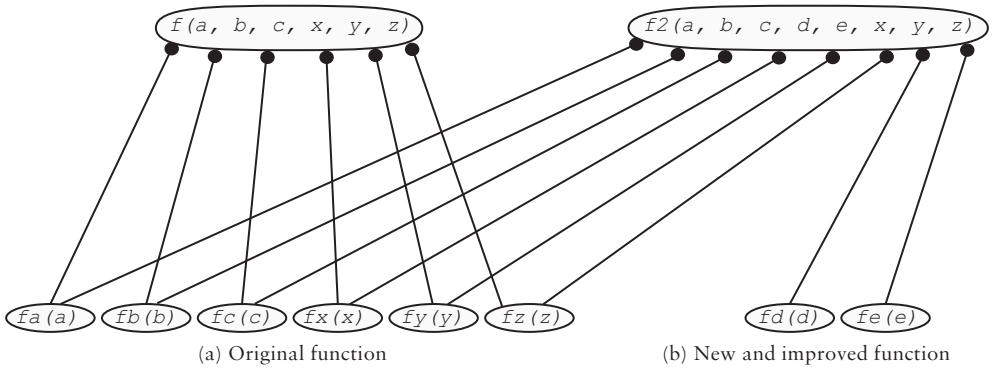


Figure 0-22: Fine-grained factoring improves stability

As a second example, suppose as part of an ultra-high-quality infrastructure we write a proprietary, high-speed parser for HTTP. Various applications begin to use this parser as part of their

---

[25] **martin95**, Chapter 3, "Metrics," pp. 246–258, specifically "(I) Instability," p. 246, and **martin03**, Chapter 20, "The Stable-Dependency Principle," pp. 261–264, specifically "(Instability I)," p. 262

[26] Throughout this book, we use the same notation (see sections 1.7.1–1.7.5) as we did in **lakos96** with a few extensions (see section 1.7.6) to support the in-structure-only nature of generic programming and concepts (which, as of C++20, are supported directly in the language).

Internet-based communication between client and server, as illustrated in Figure 0-23. The initial parser was written for HTTP version 1.0, which does not support the latest features. Over time, some clients inform us (complain) that they need the capabilities of the newer HTTP and politely request (demand) that we release a 1.1-compliant version of our HTTP parser as soon as possible (yesterday). The question now becomes, do we upgrade the original parser component (i.e., in place) or release an entirely new component that is 1.1-compliant and continue to support the original HTTP 1.0 parser component in parallel (i.e., potentially side by side in the same process)?

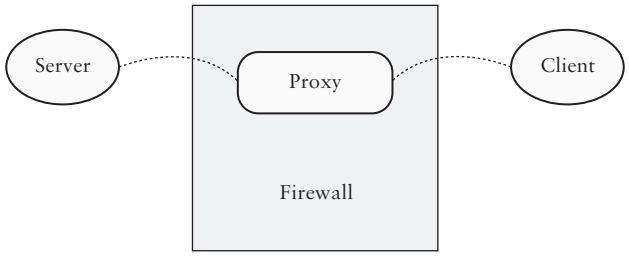


Figure 0-23: Client talking to server across firewall with proxies

The answer is that we *must* create a new component with a distinct identity and leave the original one alone! The parser is a mechanism that resides entirely within the implementation of these communication modules. For those applications that control both the client and server sides and that do not require the new features, there might never be a need to upgrade to a newer version of HTTP. Moreover, this "upgrade" is a change to existing functionality — not a bug fix and not a backward-compatible extension. As such, the decision to change properly rests entirely with each individual application owner, who cannot reasonably be required to partake in subsequent releases or otherwise be coerced according to any centralized timetable.[27]

If the proxies within the firewall require 1.1 compatibility, then exigent circumstances will oblige application developers to use the new component sooner rather than later. If, however, the proxies themselves are not yet 1.1 compatible, any such change forced upon an existing application would be catastrophic, as the newer format could not be propagated by the older

[27] There are times when library developers must coerce application developers to accept upgrades of libraries to a new release version to exploit new functionality. This coercion is especially necessary when continuous refactoring results in changes to types that are commonly used across interface boundaries (see Volume II, section 4.4). As long as the "upgrade" is substantially backward-compatible (Volume II, section 5.5), it is not unreasonable to expect that people will migrate in a "reasonable" timeframe (i.e., somewhat longer than the maximum length of a typical vacation). Even so, the precise timing of the upgrade *must* be orchestrated by the application developer. At least that way the application developer can avoid being in an incoherent state when the changeover occurs.

proxies across the current firewall. Among other things, this example should demonstrate the general need for occasionally supporting multiple versions of implementation-level functionality for simultaneous use even within a single program (see section 3.9).

Object-oriented or not, it is a cornerstone of our engineering philosophy that to test anything nontrivial *thoroughly*, we must be able to test it *hierarchically*. That is, at each level of physical hierarchy, we should be able to test each component in terms of only lower-level components, each of which has already been tested thoroughly (see section 2.14). Given that thorough testing (let alone correct implementation) of a component implies complete knowledge of the functionality upon which that component is built, any change — no matter how small — to the behavior of lower-level components potentially invalidates assumptions made by implementers at higher levels, leading to subtle defects and instability.

The concept of writing code whose behavior does not change is not new. Bertrand Meyer, author of the Eiffel programming language, describes this important aspect of a sound design as the *open-closed* principle[28,29]:

> Software should be simultaneously *open* to extension yet *closed* to modification. That is, instead of trying to capture all potential behavior within a single component, we provide the hooks necessary to extend that behavior without necessarily having to change that component's source code.

As a concrete example, let us consider the design of a **`list`** component implementing `std::list` in the C++ Standard Template Library (STL). Being that `std::list` when instantiated with a value-semantic type is itself value-semantic (see Volume II, section 4.3), the component that defines `std::list` should (ideally) also provide a free operator that implements the standard meaning of equality comparison:

```
template <class TYPE>
bool operator==(const list<TYPE>& lhs, const list<TYPE>& rhs);
    // Return 'true' if the specified 'lhs' and 'rhs' objects have
    // the same value, and 'false' otherwise.  Two 'std::list<TYPE>'
    // objects have the same value if they have the same number of
    // elements, and elements at corresponding positions have the
    // same value.
```

---

[28] **meyer95**, section 2.3, pp. 23–25

[29] See also **meyer97**, section 3.3, pp. 57–61.

Now, suppose we find that we need to compare two lists for equality in a way other than the one provided. Perhaps all that we are interested in is, say, whether the absolute values of corresponding elements are the same. What should we do?

If it were our own class and we could modify it, we might consider adding a method `absEqual` to the class, but that "solution" is entirely inappropriate because it violates the open-closed principle. We might briefly entertain providing some sort of callback or template parameter to allow us to customize the equality operation; both are ugly, however, and none of these proposed remedies guarantees a solution to other similar problems we might face. For example, suppose we also want to print the contents of the list in some specific format. Our specialized `absEqual` method will not help us.

Changing the behavior of existing reusable functionally is wrong, and even adding new behaviors to what was previously considered a complete component is usually suboptimal. In fact, any invasive remedy addressing the needs of individual clients is highly collaborative and almost always misguided. The general solution to the stability problem is to design components in such a way that they are open to efficient, noninvasive, unilateral extension by clients; in the case of the **`list`** component specifically, and containers in general, that way is to provide an iterator.[30]

---

[30] Even a `Stack` type, which traditionally provides access to only its top element, would properly have an associated iterator to support similar unilateral extensions by clients. Note that there is absolutely no issue regarding encapsulation — even with a restricted interface, clients of a `Stack` class would still be able to compare two `Stack` objects for "absolute equality" (just not efficiently).

```
bool myAreAbsEqual(const Stack& lhs, const Stack& rhs)
{
    Stack u(lhs), v(rhs);  // expensive!                          (BAD IDEA)
    while (!(u.isEmpty() || v.isEmpty())) {
        if (std::abs(u.top()) != std::abs(v.top())) {
            return false;                                        // RETURN
        }
        u.pop();
        v.pop();
    }
    return u.isEmpty() && v.isEmpty();
}
```

The iterator restores that efficiency without imposing any restriction on implementation choice or additional runtime overhead.

Providing an iterator enables each client of a container to implement virtually any appropriate extension immediately. For example, both of the functions in Figure 0-24 are independent, client-side extensions of the **`list`** component. By designing in extensibility via iterators, we can reasonably ensure that no source-code modification of the reusable container will ever be needed for that purpose. This principle of unilateral extensibility also helps to ensure that application developers will not be held hostage waiting for enhancements that can be provided only by library developers. Note that all of the container types in the C++ Standard Library come equipped with iterators that provide efficient access to every contained element.[31]

```
bool myAreAbsEqual(const List& lhs, const List& rhs)
{
    List::const_iterator lit = lhs.begin();
    List::const_iterator rit = rhs.begin();
    const List::const_iterator lend = lhs.end();
    const List::const_iterator rend = rhs.end();

    while (lit != lend && rit != rend) {
        if (std::abs(*lit) != std::abs(*rit)) {
            return false;                                          // RETURN
        }
        ++lit, ++rit;
    }
    return lend == lit && rend == rit;
}
```[32]

(a) Determining whether two lists are equivalent in a particular way

---

[31] As another less general, but much more detailed, illustration of enabling the open-closed principle, see section 3.2.8.

[32] As discussed in the preface, we will sometimes note places where a more modern dialect of C++ provides features that might better express an idea. For example, given that an iterator is a well-known *concept* having well defined syntactic and semantic properties, the specific C++ type of the iterator becomes unimportant. In such cases we might prefer to use the keyword `auto` to let the compiler figure out that type, rather than force us to type it, or even an alias for it, each time we need it, as the code itself — e.g., `lhs.begin()` — makes the *concept* clear:

```
bool myAreAbsEqual(const List& lhs, const List& rhs)
{
    auto       lit  = lhs.cbegin();
    auto       rit  = rhs.cbegin();
    auto const lend = lhs.cend();
    auto const rend = rhs.cend();

    while (lit != lend && rit != rend) {
        if (abs(*lit) != abs(*rit)) {
            return false;                                          // RETURN
        }
        ++lit, ++rit;
    }
    return lend == lit && rend == rit;   // We generally try to avoid placing modifiable operands
}                                        // on the left-hand side of an operator==.
```

```
std::ostream& myFancyPrint(std::ostream& stream, const List& object)
{
    stream << "TOP [";
    for (List::const_iterator it  = object.begin();
                              it != object.end();
                             ++it) {
        stream << ' ' << *it;
    }
    return stream << " ] BOTTOM" << std::flush;  //  useful for debugging
}[33]
```

(b) Printing the value of a list in a particular way

Figure 0-24: Unilateral client extension of operations on a `list`

Interestingly, the C++ language was originally created with a similar design goal for itself. No attempt was made to build in a complete set of useful types or mechanisms. Choosing one type or mechanism from among many competing candidates for C++ would serve to *limit* rather than *extend* its scope.

> Thus, C++ does not have built-in complex number, string, or matrix types, or direct support for concurrency, persistence, distributed computing, pattern matching, or file systems manipulation, to mention a few of the most frequently suggested extensions.[34]

Instead, the focus of the design (and evolution) of C++ was always to improve abstraction mechanisms (initially with inheritance and later with templates) to facilitate the efficient incorporation of arbitrary (user-defined) types and mechanisms supplied later in the form of well-organized, stable libraries. Such libraries can provide several concrete variants of any abstract type or mechanism, which could exist concurrently, even within a single program. Much of this book addresses extending this noble architectural philosophy — rooted in the language itself — toward its natural conclusion in the form of well-factored, enterprise-wide

---

[33] In this case, we might prefer a range-based `for` loop along with `auto` (introduced as part of C++11):

```
std::ostream& myFancyPrint(std::ostream& stream, const List& object)
{
    stream << "TOP [";
    for (const auto &v : object) {
        stream << ' ' << v;
    }
    return stream << " ] BOTTOM";  // '<< std::flush' is optional.
}
```

[34] **stroustrup94**, section 2.1, pp. 27–29, specifically the middle of p. 28

library-based solutions (see section 0.9). Moreover, C++ allows you to create these libraries and new types in such a way that their interfaces are the same as, and behave as if they were, first-class citizens like `int`.

Inheritance, used properly (Volume II, section 4.6), is arguably the most powerful mechanism for enabling, on a large scale, extension without modification. Consider the simple report-generator architecture of Figure 0-25. In this classical *layered* design (see section 3.7.2), the `PostscriptOutputDevice` module takes output requests and translates them into a form suitable for driving a PostScript printer. All of the detailed knowledge regarding writing accounts properly resides in the `AccountReportWriter` module (and not the `PostscriptOutputDevice` module). So far so good.

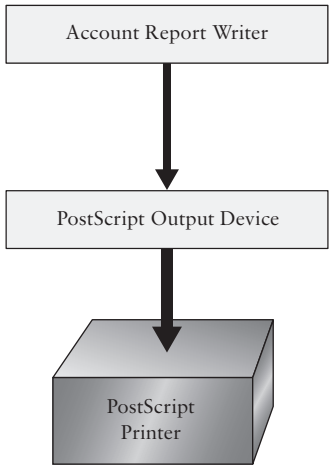


Figure 0-25: A simple `Account` report generator

We can extend the kinds of reports we can produce simply by adding new report modules, as shown in Figure 0-26. And, assuming careful fine-grained factoring, much of the code used to implement the original `AccountReportWriter` module will be common to these new report writers.

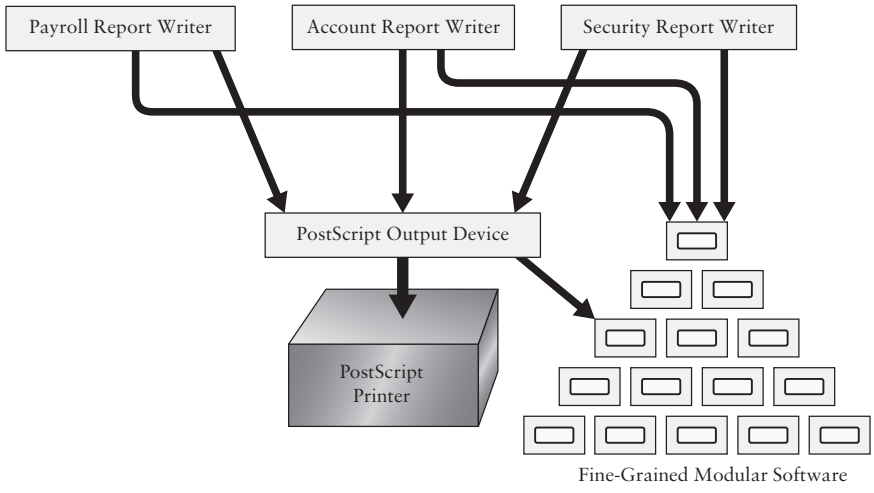


Figure 0-26: Adding new kinds of reports

But what happens when we want to add new kinds of output devices? How are we supposed to extend that architecture without modifying it? The expedient solution is to duplicate and rename the entire suite of reports and retarget them to the new output device as in Figure 0-27. Needless to say, from a maintenance perspective, this is not a good, long-term solution.

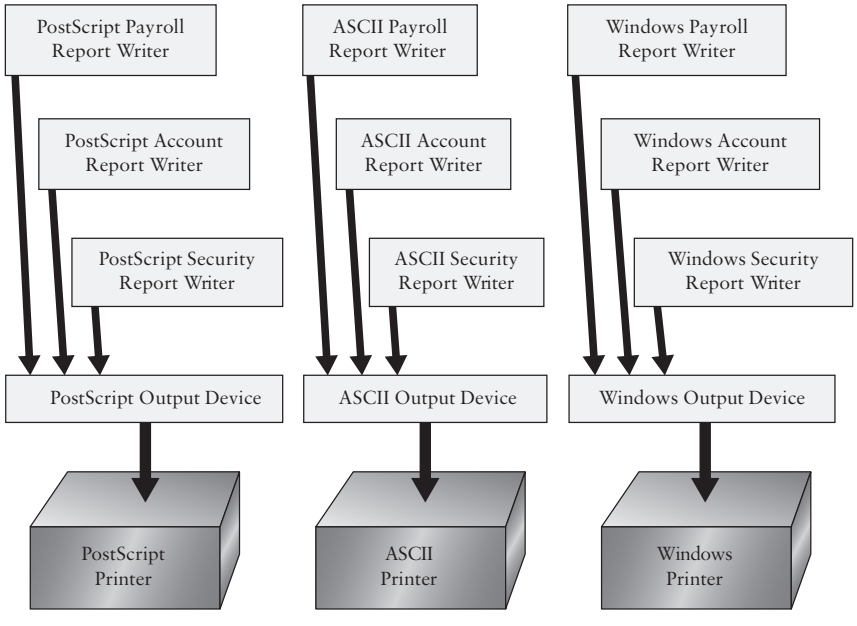


Figure 0-27: Copying code to add new kinds of output devices (BAD IDEA)

Suppose instead we decide to consolidate all of the device interfaces into one interface that can then be used to drive all appropriate devices. Given a collection of existing interfaces, we might consider making this component a wrapper that logically encapsulates the specific devices, as shown in Figure 0-28. This solution, although often employed in practice, also fails on multiple counts. Apart from having to modify the source of the output manager every time a new device comes online, we impose an outrageous physical dependency: *Every* report writer, to be used (or even tested), must link with *every* known output device. As if that were not bad enough, some devices (e.g., the Windows output device) might not exist on every supported platform! Enough said.

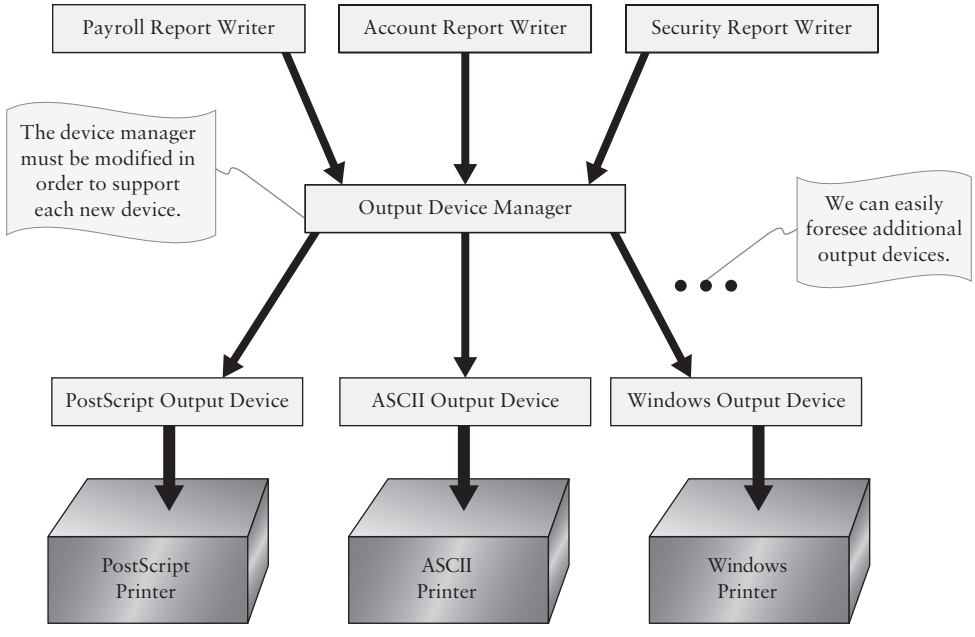


Figure 0-28: Encapsulating a readily extensible collection (BAD IDEA)

As will be a recurring theme throughout this book, we will choose to avoid encapsulating inherently extensible, nonportable, third-party, or otherwise uncontrollable or risky software that could potentially produce a destabilizing effect on our core libraries. Instead, we will capture the definition of such behavior — typically[35] as a *protocol* (pure abstract interface) class (see section 1.7.5) — along with detailed comments delineating precisely what behavior is expected. In this way, new output devices may be introduced by applications at will; the only caveat is that these new concrete implementations satisfy the *contract* (see Volume II,

[35] Section 3.5.7 explores protocols (see section 3.5.7.4) along with several other alternatives including concepts (see section 3.5.7.5).

section 5.2) imposed by the abstract interface. In this design, both clients and implementations co-exist as peers, each depending on only the lower-level component containing the interface. This abstract-interface approach has the further benefit of facilitating client testing through the idiomatic reuse of a ("mock") `TestOutputDevice` (see Volume II, Chapter 4, and Volume III, Chapter 9). A fully factored, bilaterally extensible, yet absolutely stable solution to our report-writing problem is provided in Figure 0-29.

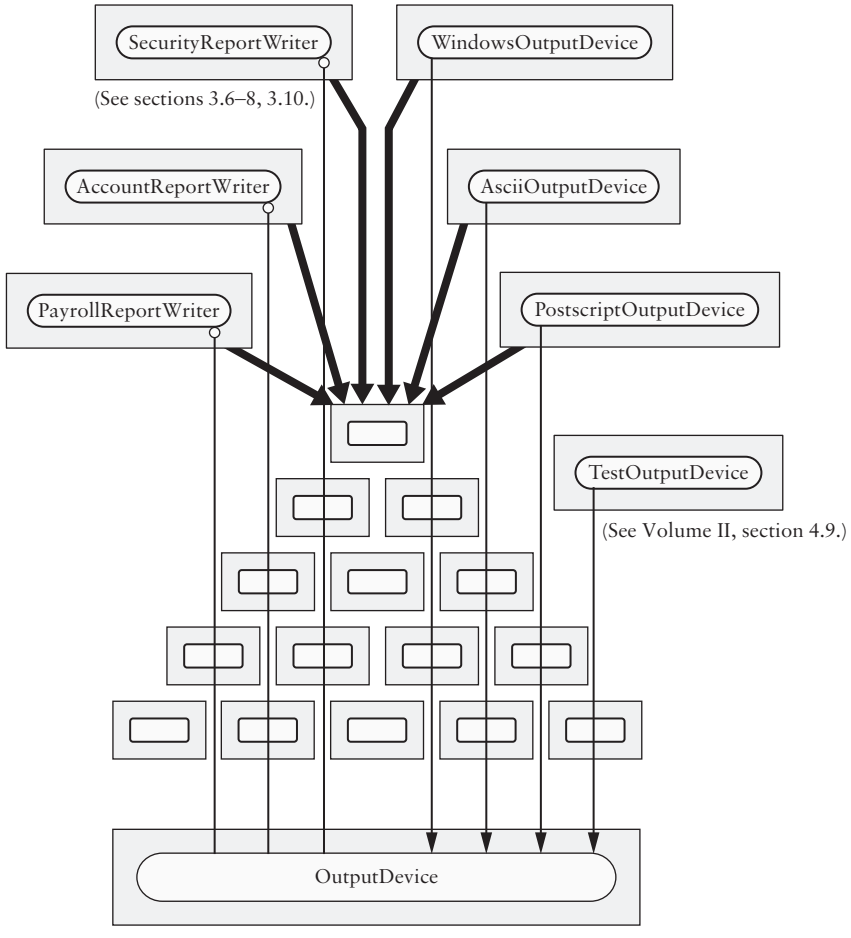


Figure 0-29: A fully factored solution

Not all software — particularly at the application level — can be stable. The behavior of each version of an application must in some way be different from the last or (apart from pure performance tuning) what would be the point? When developing high-level application code, it might be more effective to drop stability and concentrate on malleability. That is, instead of attempting

to apply the open-closed principle, we accept that the code will change and focus on making such changes safe and easy. For example, it is a common practice to group code that is likely to change together in close proximity (see section 3.3.6); ideally, the amount of co-dependent code would be small enough to fit within a single component. However, when incompatible changes to existing behavior are permitted to be exposed through public interfaces, that body of software is *not* stable. Whether or not such code is considered malleable, it is definitely not reusable.

As Figure 0-30 suggests, how malleable versus stable software is used is quite different. Perhaps the most salient characteristic distinguishing malleable software from stable software is that malleable software must have exactly one master that dictates changes in its behavior (Figure 0-30a), whereas reusable software — once created — has none (Figure 0-30b). Since the number of clients that may depend on a reusable component to do what it does now is unbounded, none (i.e., no proper subset) of them may reasonably require an incompatible change.

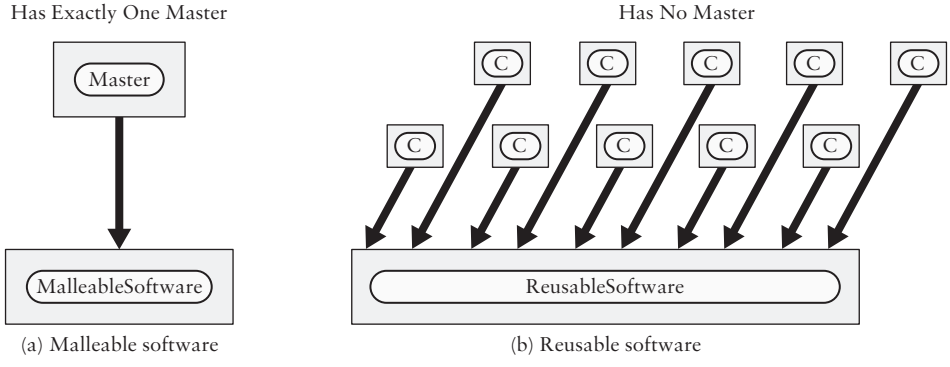


Figure 0-30: Malleable versus reusable software

If the established behavior of software must change, then it should do so at the behest of (and affect only) a single master — i.e., a single application. The corollary is that malleable software belonging to one application must not be shared with another (see section 2.13). As Figure 0-31 illustrates, allowing more than one master the authority to unilaterally require changes to shared software leads to an over-constrained problem (Figure 0-31a), often with suboptimal results (Figure 0-31b). Even when existing logical behavior is preserved, modifications that adversely affect runtime performance or other physical properties, such as link-time dependencies, portability, and so on, can critically impact stability. To achieve effective hierarchical reuse, it is imperative that, with only rare exceptions, all aspects of a component's contract — expressed or implied — continue to be honored for the life of the component (see Volume II, section 5.5).

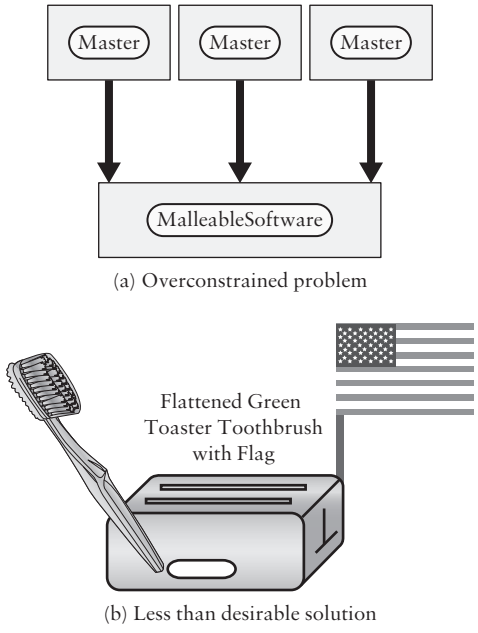


Figure 0-31: Software having multiple masters (BAD IDEA)

Infrastructure, once in use, must be maintained and enhanced. With conventional, coarse-grained factoring, the more this infrastructure software is used, the more expensive it becomes to maintain as eager users clamor for more and more enhancements. Failure to provide the enhancements leaves users feeling unsupported and unhappy, especially since there is typically no way for them to unilaterally extend that functionality themselves — a violation of the open-closed principle.

In some organizations, this increasing cost will not be sustainable, and the infrastructure group, which might not be perceived as contributing directly to the bottom line, would be the first to be sacrificed. By ensuring a fine-grained rendering of reusable infrastructure, many of the costs associated with supporting reuse are eliminated. Individual components remain stable, and any enhancements become additive instead of iterative.

To summarize, the way we achieve effective reuse is through *hierarchical* reuse. The frequency with which combinations of low, intermediate, and high-level reuse *can* occur is staggering, which we will illustrate by analogy in section 0.8. Examples of commonly reusable yet very low-level pieces include memory pools and allocators, managed/shared pointers, functors,

scoped guards, and, of course, synchronization and communication primitives, just to name a few. Throughout this book, we will demonstrate the process of discovering, designing, implementing, and testing these low-level, yet highly leverageable components. Such meticulous factoring is not easy; yet, it is precisely this careful attention to fine-grained factoring that helps achieve unparalleled reuse and stability by allowing us to compose, and thereby extend, the functionality provided by our software infrastructure without always having to change it.

So far, we have characterized the software we advocate as *finely graduated* and *granular*, of having a clean decomposition, and of generally allowing only common, simple (e.g., *scalar*) types to pass through (well-documented) functional interfaces. These simple types form the basic vocabulary (see Volume II, section 4.4) by which information is transmitted across function boundaries. We have argued that interfaces having a relatively small *surface area* that are easy to describe are fundamentally more able and likely to be reused in new contexts than more complicated or *collaborative* ones.

In this section, we introduced a new dimension by which we characterize software: stability. To the extent possible, we prefer our software — particularly our library software — to be stable, and therefore potentially reusable. Designing for stability (see Volume II, section 5.6) is a staple of our daily development work. In fact, our general approach to large-scale software development fundamentally requires the construction of new software by recursively building on what we have rather than iteratively changing it.

The architectural analogy to pure functional languages (e.g., lambda calculus or pure Lisp) rather than imperative languages (such as C or Java) is also strikingly similar to the fascinating compile-time style employed by templates and metafunctions in modern generic programming using C++[36] (see Volume II, section 4.5). That is, once we "instantiate" a reusable component (i.e., write and release it), we can refer to (i.e., depend on) it, but we cannot subsequently change its "value" (i.e., change its defined or implied logical contract or substantially degrade any aspect of its essential physical characteristics). Such is how a firm-wide repository of hierarchically reusable software that is not only ultra-high quality but also ultra-stable accumulates (see section 0.9). In the next section, we will begin to take a closer look at the *physical* aspects of finely graduated, granular software.

---

[36] **alexandrescu01**, section 3.5, p. 55

0.6 The Key Role of Physical Design

Developing successful software on a large scale demands a thorough understanding of two distinct but highly interrelated aspects of design: *logical* design and *physical* design. Logical design, as we will use the term, addresses all *functional* aspects of the software we develop. A functional specification, for example, governs the overall logical design of an application, subsystem, or individual component. The decision to make a particular class concrete or instead provide an abstract interface for the needed functionality is a fairly high-level logical design choice. Whether to use blocking or nonblocking transport (i.e., interprocess communication) with or without threads would also fall under the umbrella of logical design.

Historically, most books on C++ addressed *only* logical design and often at a very low level. For example, whether a particular class should or should not have a copy constructor is a low-level yet very important logical design issue. Deliberately making a particular operator (e.g., `operator==`) a *free* (i.e., nonmember) function is also a low-level logical design choice. Even selecting the specific private data members of a class would be considered part of low-level logical design. However, logical design alone on any level fails to consider many important practical aspects of software development. As development organizations get larger, forces of a different nature come into play.

Physical design, as defined in Figure 0-32, addresses issues surrounding the placement of logical entities, such as classes and functions, into physical ones, such as files and libraries. All design has a physical aspect. That is because all of the source code that makes up a typical C++ program resides in files, which are physical. When we talk about reuse, for example, we might be thinking about a subroutine or class, but such things cannot be reused directly. What can potentially be reused is the set of files that define and implement that logical content. Hence, the way in which we aggregate our logical constructs into discrete physical modules is an essential aspect of design for reuse, in particular, and sound design, in general.

Physical Design

1. *n*. The arrangement of source code within files and files within libraries
2. *v.tr*. To partition source code among files and files among libraries

Figure 0-32: Definition of physical design

Physical design dictates *where* (physically) — e.g., in what library — codified functionality belongs relative to other functionality throughout our enterprise. Recognizing compile-time and link-time dependencies across physical boundaries will help shape both logical and physical design aspects of our software. For example, common functionality intended for firm-wide reuse belongs (physically) at a *low level* in our software repository, where application software developers working at *higher levels* can find and use it (e.g., see Figure 0-7b, section 0.2), independently of other applications.

It has become well-recognized[37,38,39] that *cyclic* dependencies among (physical) modules are to be avoided (see sections 2.2–2.4, 2.6, 2.8–2.9, 2.14, and 3.4). By employing established physical design techniques (see section 3.5), we ensure that we pay (in terms of link time and executable size) only for what we need (see section 3.6). Moreover, we want to develop reusable solutions that are *independent* of peer solutions (see section 3.7) and can co-exist with them in the same process (see section 3.9). Detailed physical design will require us to answer correctly many important questions that are simply not addressed by logical design. For example,

- Should two given classes be defined in the same header file or in two separate ones? (See section 3.3.)
- Should a given translation unit depend on some other translation unit at all? (See section 3.8.)
- Should we `#include` the header file containing the declaration of a given logical entity or instead simply *forward declare* it? (See Volume II, section 6.6.)
- Should a given function be declared `inline`? (See Volume II, section 6.7.)

These kinds of physical design questions, while closely tied to the technical aspects of development, bring with them a dimension of design with which even "expert" software developers might have little or no experience. *Physical design*, though less widely understood than *logical design*, nonetheless plays an increasingly crucial role as the number and size of the applications and libraries developed within a group, department, or company grow.

Combining thoughtful logical design with careful physical packaging will enable software developers to "mix and match" existing solutions, subsolutions, and so on, at finely graduated levels of integration (section 0.3). This combinatorial flexibility along with logical and physical

---

[37] **lakos96**, section 4.11, pp. 184–187, and section 7.3, pp. 493–503

[38] **sutter05**, item 22, pp. 40–41

[39] With respect to compile-time dependency (only), see also **meyers05**, item 31, pp. 140–148.

interoperability goes a long way toward maximizing the economies of scale made possible through *hierarchical reuse* (section 0.4).

We will return to the topic of large-scale physical design in Chapter 3, after we have presented motivation (this chapter), discussed some basic ideas involving compilers and linkers (Chapter 1), and learned how logical content can be colocated, packaged, and aggregated effectively (Chapter 2).

0.7 Physically Uniform Software: The Component

The physical form in which our software is cast plays a vital role in our ability to manage it. Apart from having a sound physical structure, requiring a *physically uniform* organization for our proprietary software greatly enhances the ability of ourselves and others to understand, use, and maintain it. Moreover, physical uniformity encourages the creation of effective development tools (e.g., rule checkers, dependency analyzers, documentation extractors) that operate on the software directly (as data). "Foolish consistency [might well be] the hobgoblin of little minds," but we're quite sure Emerson[40] was not referring to the physical form of software when he wrote that.

To motivate the importance of *physical uniformity* in software, let us take a moment to consider other forms of physical media such as video cassette tapes, compact discs, DVDs, etc. The logical content of these entities, apart from quantity, is not inherently restricted. The same physical cartridge you might rent from your local video store could contain Spielberg's last hit, a travel documentary, or even a workout video. The logical content could be anything, but the physical form is the same.

Some of us might recall that in the early days of video cassettes there were two incompatible formats, VHS and Beta. *Which was better* was not nearly as important as that *there were two*! Soon there was one! The problem resurfaced when movies became available on DVDs. Having both tapes and DVDs for video was also redundant. Renting a video requires not only finding the desired movie but also ensuring compatible equipment. The more physical forms, the more onerous the problem with interoperability.

Adopting a standard physical cartridge to represent logical content greatly simplifies the integration of supporting tools. A DVD burner, for example, enjoys economies of scale because it

---

[40] In his essay "Self-Reliance," R. W. Emerson "does not explain the difference between foolish and wise consistency" (**hirsch02**, "Proverbs," pp. 47–58, specifically p. 51).

will work on all DVDs from all manufacturers (independent of content). In contrast, imagine if each movie title came on its own physically unique medium — every title would require its own set of equipment to view and maintain it!

The same arguments for interoperability across physical devices apply to software modules. Requiring one standard physical format to hold all atomic units of software similarly facilitates the creation of a more supportive development environment. Simply knowing that tools enlisted to support one project will also be viable on the next makes their development (or procurement) more cost effective and therefore more likely. Time and time again, uniformity of this kind has invariably shown to lead to a more repeatable, predictable development process.

Consistency of physical form also brings a welcome degree of familiarity to human beings. Many people know how to view a DVD. When applied in new contexts, say, when enrolled in a video course, students would not have to learn new organizational skills to view their lectures. Analogously, a high degree of physical uniformity in software allows engineers to circulate more readily among different projects. In software, the term of art is *developer mobility*. Be it video courses or software, physical uniformity makes it easier for people to focus more of their energy on the logical content, which, after all, is the goal.

For our proprietary software to be fully interoperable across all applications, its physical form and organization must be entirely independent of the functionality that it embodies. We cannot know precisely what applications we are going to develop now — let alone in the future. However, the physical organization we are about to describe has demonstrated over time that it serves well in industries as diverse as computer-aided design and financial services. Moreover, this hierarchical *physical* structure is inherently scalable, having successfully addressed systems embodying source code with line counts ranging from less than 10,000 to more than 100,000,000. The most fundamental organizational unit in our component-based methodology is, not surprisingly, called a *component*, which we will introduce here momentarily and, in Chapter 1, explore thoroughly. Later, in Chapter 2, we will describe a global *meta*framework[41] that makes all these components play together.

The term *component* has come to mean different things in different contexts. Many system architects consider a component to be a *unit of deployment*, such as a COM component or Active X control (*plug-in*). In this sense, a `main` program (e.g., a server) could be a "component" in a distributed network of separately running executables (processes). Even an entire library could be

[41] A framework (a.k.a. an application framework) is typically subject (domain) specific; a *meta*framework — i.e., a framework to build frameworks — is not.

considered just a single "component" of a large development environment. Others, focusing more on individual programs, might view a component as any significant subsystem, such as an order book, a router, or a logger. Still others view a component as any discrete piece of logical content such as a class or suite of `static` methods nested within a (*utility*) `struct` (or, equivalently, free functions nested within a local namespace). Although each of these interpretations is acceptable in some appropriate context, none of them reflects what *we* mean by the term *component*.

By our definition, a *component* is the *atomic unit* of physical design. In C++, it takes the form of a `.h`/`.cpp` pair (see section 1.6).[42] Associated with each component should be a standalone test driver (see Volume III, section 7.5) capable of exercising all functionality implemented within that component. This ubiquitous physical organizational pattern, illustrated schematically in Figure 0-33, is the basic unit of packaging for *all* of our application and library software.

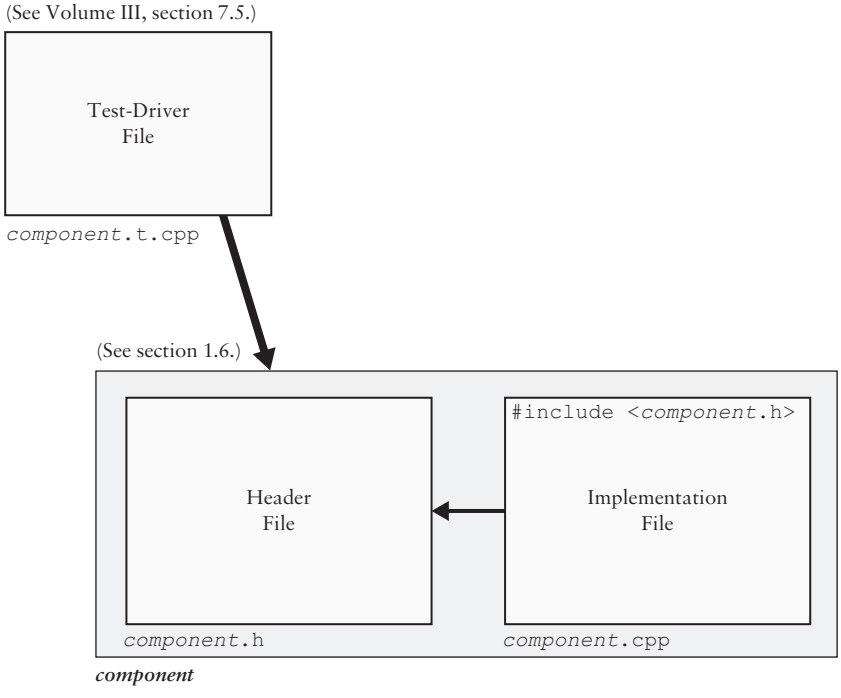


Figure 0-33: Schematic view of a component and its test driver

[42] We use the suffix `.cpp` to distinguish C++ source files from C source files. We will use `.h` and `.cpp` consistently throughout this book.

A component embodies a relatively *small* amount of *logical design*; its contents are cohesive in that the functionality provided naturally belongs together. We say that a component is *atomic* because (1) use of any part potentially implies use of the whole and (2) any explicit desire to use only part of a component is a strong indication that the functionality supplied by the component is insufficiently factored. Lastly, a component has a *well-defined* (and, ideally, *well-designed*) interface (e.g., it should be designed to be testable).

As a concrete example, consider the header file shown in Figure 0-34 for a "toy" stack component, **`my_stack`**, implementing a basic integer stack abstraction. A `Stack` is a kind of container. Recall from section 0.5 (and our discussion of the open-closed principle) that containers should have iterators, so it follows that our `Stack` class should have an iterator as well. Although access to other than the top element of a stack is not normally thought of as part of a stack abstraction, providing an iterator makes the component more generally extensible by clients and therefore more stable.

From this header file, we can see that the component contains two distinct classes, `Stack` and `StackConstIterator`. We can also see that there are two free (i.e., nonmember) operator functions implementing `==` and `!=` between two objects of type `StackConstIterator`, as well as between two `Stack` objects. Finally, because a `Stack` represents a *value* that is meaningful outside of the current process (see Volume II, section 4.1), we have included an output operator `<<` to render this *value* in some human-readable form.[43] Note that, for the purposes of exposition, we will defer any discussion of packaging (e.g., file prefixes and namespaces) until Chapter 2.

[43] In practice, we might also want to supply methods, such as `streamIn` and `streamOut`, to serialize this value to an abstract byte stream (see Volume II, section 4.1). Note that we have also provided for an optional memory allocator, supplied at construction, to be used to manage dynamically allocated memory throughout the lifetime of a stack object (see Volume II, section 4.10).

```
// my_stack.h                                              (ignoring packages for now)
#ifndef INCLUDED_MY_STACK   // internal include guard
#define INCLUDED_MY_STACK                                                (See section 1.5.)

#include <iosfwd>

// ...

namespace bslma { class Allocator; }[44]                          (See Volume II, section 4.10.)

// ...                  (ignoring other namespaces for now)

class StackConstIterator {   // const forward iterator
    // ...
    friend class Stack;   // single unit of encapsulation
  public:
    // ...
};

// FREE OPERATORS
bool operator==(const StackConstIterator& lhs, const StackConstIterator& rhs);
    // Return 'true' if the specified 'lhs' and 'rhs' iterators refer to the
    // same element of the same stack, and 'false' otherwise.

bool operator!=(const StackConstIterator& lhs, const StackConstIterator& rhs);
    // Return 'true' if the specified 'lhs' and 'rhs' iterators do not refer
    // to the same element of the same stack, and 'false' otherwise.

// ...

class Stack {
    // This class implements a "toy" value-semantic integer stack class...

    // DATA
    int              *d_stack_p;       // dynamically allocated array
    std::size_t       d_capacity;      // capacity of dynamic array
    vsize_t           d_length;        // length of stack (and stack pointer)
    bslma::Allocator *d_allocator_p;[45]  // memory allocator; held, but not owned
                                                          (See Volume II, section 4.10.)
  public:
    // TYPES
    typedef StackConstIterator const_iterator;   // typical STL-style iterator
        // Alias for a standard forward iterator.

    // CREATORS
    Stack(bslma::Allocator *basicAllocator = 0);
        // Create an empty 'Stack' object.  Optionally specify a
        // 'basicAllocator' used to supply memory.  If 'basicAlloctor' is 0,
        // the currently installed default allocator is used.
```

(continues)

[44] **bde14,** subdirectory `/groups/bsl/bslma/`

[45] **bde14**, subdirectory `/groups/bsl/bslma/`

(continued)

```
    Stack(const Stack& original, bslma::Allocator *basicAllocator = 0);
        // Create a 'Stack' object having the value of the specified 'original'
        // object.  Optionally specify a 'basicAllocator used to supply memory.
        // If 'basicAllocator' is 0, the currently installed default allocator
        // is used.

    ~Stack();
        // Destroy this object.

    // MANIPULATORS
    Stack& operator=(const Stack& rhs);
        // Assign to this object the value of the specified 'rhs' object, and
        // return a reference providing modifiable access to this object.

    void push(int value);
        // Append the specified 'value' to the top of this stack.

    void pop();
        // Remove the value at the top of this stack.  The behavior is
        // undefined unless 'isEmpty()' returns 'false'.

    // ...                (ignoring "aspects" for now)

    // ACCESSORS
    const_iterator begin() const;
        // Return a standard forward iterator referring to the top element
        // of this stack, or 'end()' if this stack is empty.

    const_iterator end() const;
        // Return an iterator indicating the "one-past-the-end" position in
        // this stack.

    bool isEmpty() const;
        // Return 'true' if the number of elements in this stack is 0, and
        // 'false' otherwise.

    const int& top() const;
        // Return a 'const' reference to the top element of this stack.
        // The behavior is undefined unless 'isEmpty()' returns 'false'.

    // ...                (ignoring "aspects" for now)

};

// FREE OPERATORS
bool operator==(const Stack& lhs, const Stack& rhs);
    // Return 'true' if the specified 'lhs' and 'rhs' objects have the same
    // value, and 'false' otherwise.  Two 'Stack' objects have the same value
    // if they have the same number of elements, and corresponding elements
    // have the same value.

bool operator!=(const Stack& lhs, const Stack& rhs);
    // Return 'true' if the specified 'lhs' and 'rhs' objects do not have the
    // same value, and 'false' otherwise.  Two 'Stack' objects do not have the
```

(continues)

(continued)

```
    // same value if they do not have the same number of elements, or any
    // corresponding elements do not have the same value.

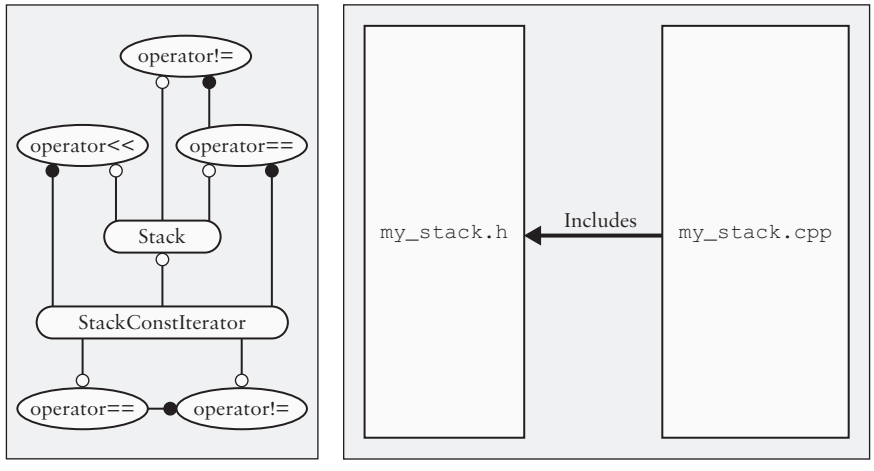
std::ostream operator<<(std::ostream& stream, Stack& stack);
    // Write the value of the specified 'stack' to the specified output
    // 'stream' in some single-line, human-readable format...

// ...
// ... (ignoring inline [member/free-operator] function definitions for now)
// ...

#endif
```

Figure 0-34: Header file `my_stack.h` for a toy (integer) stack component

Peeking at the implementation, we would discover that both `operator==` and `operator<<` for `Stack` use class `StackConstIterator` and that `operator!=` for both classes is implemented in terms of the corresponding `operator==`. The complete set of logical entities at file scope in component **`my_stack`** is pictured in Figure 0-35a using the notation of section 1.7. The physical entities (`my_stack.h` and `my_stack.cpp`) along with their canonical physical relationship (see section 1.6.1) are depicted in Figure 0-35b.



(a) Logical view (b) Physical view

Figure 0-35: Two views of our `my_stack` component

More generally, a component will typically define one or more closely related classes and any free operators deemed appropriate for the abstraction it supports. As Figure 0-36 demonstrates, the component name will be reflected in every top-level logical entity it contains (see section 2.4.9). Basic types (e.g., `Point`, `Datetime`, `BigInt`, `ScopedGuard`) will each be implemented in a component containing a single class (Figure 0-36a). Generic container classes (e.g., `Set`, `Stack`, `List`) will typically be implemented in a component containing the principle class and its iterator(s) (Figure 0-36b). More complex abstractions involving multiple types (e.g., `Graph`, `Schema`, `ConcreteWidgetFactory`) can embody several classes in a single component (Figure 0-36c). Finally, classes that provide a wrapper (a.k.a. a *facade*) for an entire subsystem (e.g., `Simulator`, `XmlParser`, `MatchingEngine`) may form a thin encapsulating layer consisting of multiple principle classes and many iterators (Figure 0-36d). As a rule, however, we place each class in a separate component to avoid gratuitous physical coupling — that is, unless there is a compelling engineering reason to do otherwise. We address specific design criteria justifying colocation in section 3.3 (see Figure 3-20, section 3.3.1).[46,47]

---

[46] Note that, along these lines, if an underscore is used — e.g., `Graph_Node`, `Graph_NodeIterator` — one could keep an unambiguous correspondence between the class name and the component name. We almost fully achieve this goal (see section 2.4.6) but not quite as we have chosen to utilize the precious "underscore" identifier character (_) for something even more pressing, component-private classes (see section 2.7.3).

[47] Note that this figure differs from a similar one (**lakos96**, section 3.1, Figure 3-1, p. 102) for three reasons: (1) we now use logical package namespaces (represented here as `my::`) as opposed to prefixes such as `my_` (see section 2.4.6); (2) naming requirements are now such that the lower-cased name of every logical entity, apart from operator and aspect functions (see Volume II, section 6.13), must have the base name (see section 2.4.7) of the component as a prefix (see section 2.4.8); and (3) the iterator model (STL-style) now has the iterator used in the interface of the container (see Figure 2-54, section 2.7.4) instead of vice versa (lakos96-style), along with a renewed passion to avoid cyclic logical relationships — even within a single component!

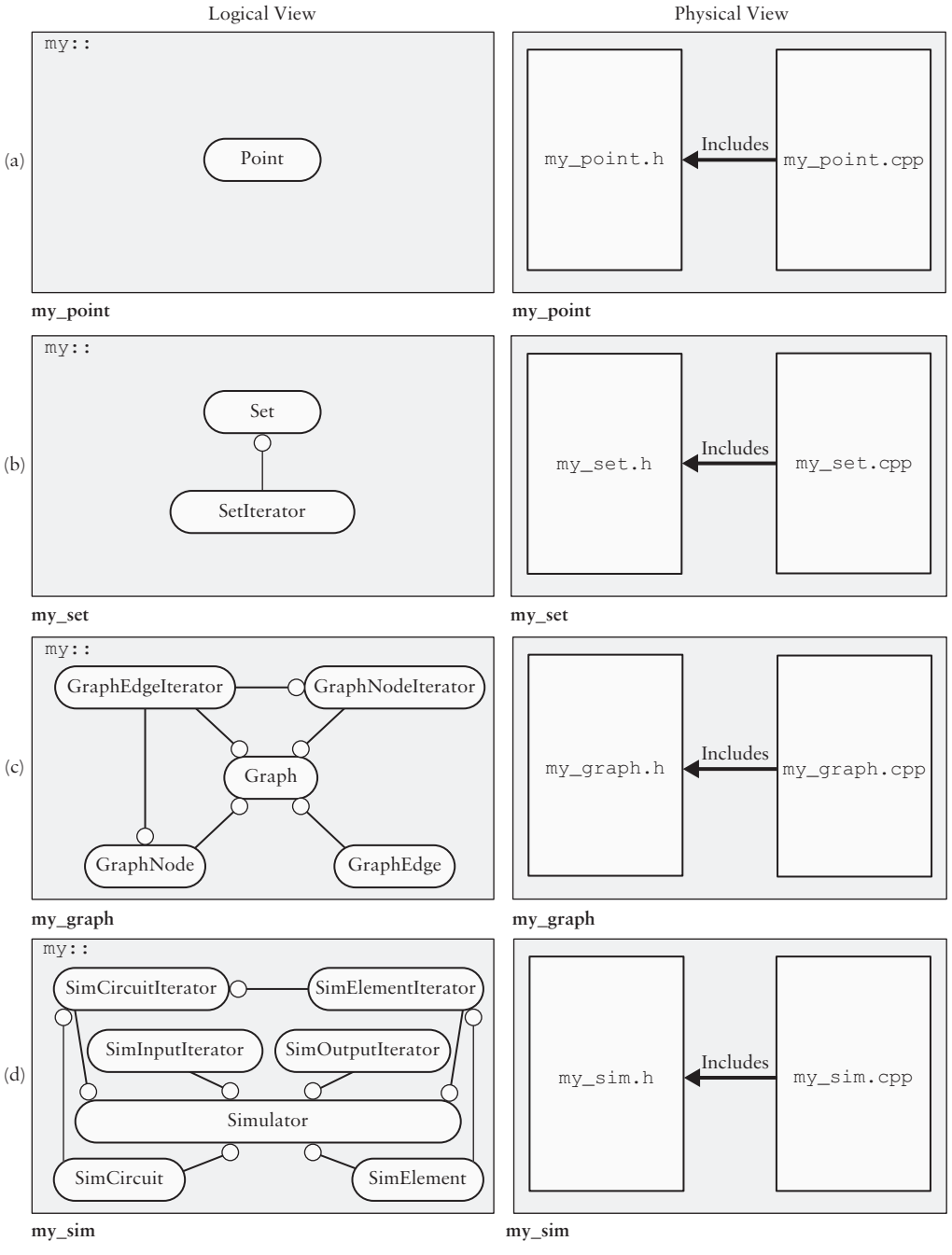


Figure 0-36: Logical versus physical view of several components

Each of the components illustrated in Figure 0-36 (like every other component) has a physical as well as a logical view. The physical view consists of the `.h` file and the `.cpp` file, with the `.h` file included (as the first substantive line) in the `.cpp` file (see section 1.6.1). Moreover, the *size* of a component, as measured by both implementation complexity and lines of source, will turn out to be relatively constant (see section 2.2.2). From a physical perspective, these components are more similar than different.

From a logical perspective, however, components such as the one shown in Figure 0-36d clearly *do* a lot more than those like the one in Figure 0-36a. Although it is plausible that much if not most[48] of the implementation of a **`point`** component would be embodied directly in that component's source, such is generally not the case. The implementation of a map, for example, will likely delegate part of its functionality to a few lower-level components, such as a (default) memory allocator (see Volume II, section 4.10), along with various traits and metafunctions. And, as Figure 0-37 illustrates, a more substantial piece of machinery, such as the matching engine for a trading system, will delegate to layers upon layers of subfunctionality, all of which would ideally be thoughtfully factored and placed in components of comparable size, distributed appropriately throughout our firm's repository of hierarchically reusable software (see section 0.9).

[48] But probably not all. Consider that a point may have associated traits such as "bitwise copyable" that it would want to include from another component. Output and serialization are two other operations that would likely depend on other components. Finally, *defensive* checks (see Volume II, section 6.8) for *narrow* contracts (see Volume II, section 5.3) would also best be imported from a separate low-level component.

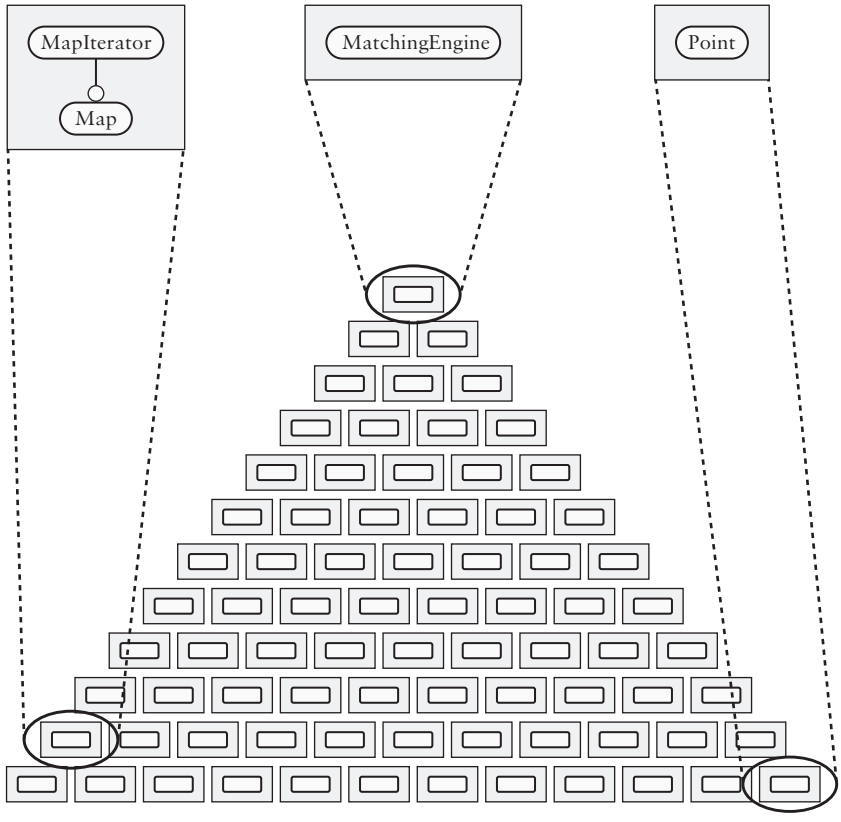


Figure 0-37: Physical location — *not* size — governs a component's scope.

It is worth noting that this view of composition — i.e., via recursive delegation to stable, reusable components — is fundamentally different from the classical nested, collaborative, and therefore private subarchitectures typical in traditional application development. In that design paradigm, systems own and isolate their subsystems, which in turn own and isolate theirs — the advantage being that the subsystems are not otherwise accessible and therefore need not be stable. The disadvantage is that there is no sharing of common subsolutions across subsystems — in other words, no reuse.

In this new compositional paradigm, the component implementing the matching engine serves as an encapsulating wrapper[49] (see Figure 3-5, section 3.1.10). Only those components that implement the (vocabulary) types (Volume II, section 4.4) that flow in and out of the wrapper

[49] **lakos96**, section 5.10, pp. 312–324, specifically Figure 5-95, p. 319

component itself participate in its contract (Volume II, section 5.2). Dependency on all other objects are implementation details that are not discoverable programmatically. With respect to these implementation-only components, encapsulating a component means encapsulating its *use* and not the component itself[50] (see section 3.5.10 and also Figure 3-140, section 3.11.2). This fundamentally different form of composition and encapsulation — i.e., based on dependency and not containment — is what makes hierarchical reuse (section 0.4) of stable subsolutions (section 0.5) possible.

As described here, a component serves as the appropriate fundamental unit of both logical and physical design. Components, like classes, allow for factoring, thus helping us to make small problems out of big ones, and we can solve small problems well. But components, unlike classes, enable consideration of objective physical properties (e.g., compile- and link-time dependencies) that are beyond the scope of logical design alone. Components, being physical entities, unite intimately related classes (e.g., a container and its iterators), along with closely related free operators (e.g., the output and equality operators, `operator<<` and `operator==`, respectively), in a single cohesive unit. Components, being represented in terms of files (as opposed to regions of source code), are physical modules that can be lifted individually from one *platform*[51] and deployed on another without ever having to invoke a text editor.[52] Finally, well-crafted components can be tested thoroughly because they are small and were designed with testing in mind (see Volume II, section 4.9). As we will see, the component — as *we* define it — enables us to realize the enormous synergy between good logical and physical design.

0.8 Quantifying Hierarchical Reuse: An Analogy

The business of top-down design of application software is, in effect, a series of partitioning problems that optimizes a complex global cost function. Functionality, ease of use, availability, reliability, maintainability, portability, time to market, and, of course, total development cost (to name just a few) are all potential parameters to the problem. We have also discussed the bottom-up approach of accumulating and maintaining a fine-grained hierarchy of useful and potentially reusable (stable) library solutions to expedite future application development.

---

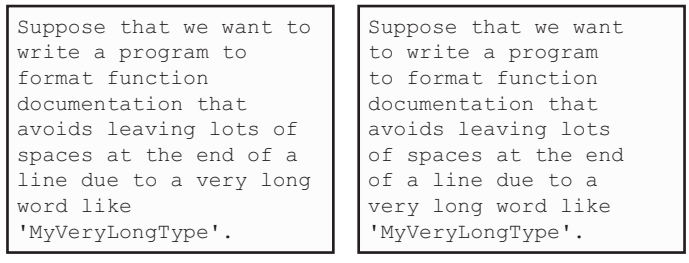
[50] **lakos96**, section 5.10, pp. 312–324, specifically the DEFINITION on p. 318

[51] Throughout this book we will use the term *platform* loosely (but consistently) to connote a distinct combination of underlying hardware, operating system, and compiler used to build and run software such that object code compiled on any one platform is typically usable on any other.

[52] I.e., without having to physically modify source code.

So far, we have only alluded to the mathematical inclination for the scalability of our general approach, namely, the well-reasoned hierarchical reuse of component-based software. To complete the task of assessing the enormous potential benefits of hierarchical reuse in software development, we will demonstrate them quantitatively using an analogous, but far simpler, much more tractable problem.

Suppose that we want to write a program to format function documentation that avoids leaving lots of spaces at the end of a line due to a long word like `MyVeryLongType`. Figure 0-38a illustrates documentation that is formatted by placing as many words as will fit on a line before advancing to the next. Figure 0-38b illustrates a more aesthetically pleasing rendering of the same text with a less-ragged right edge. The aesthetic solution is more computationally intensive because, instead of a straightforward linear partitioning algorithm, we now have to optimize a nontrivial global cost function, much as we do in partitioning software during classical top-down design.



(a) Local optimization (b) Global optimization

Figure 0-38: Simple text-filling problem with different cost functions

More formally, we are trying to solve the following "hard" problem:

> Given (1) a maximum line length L and (2) a sequence of N words $\{w_0 \ldots w_{N-1}\}$, each having an independent length of from 1 to L characters, partition the words such that the sum of the results of applying some arbitrary, non-negatively valued, monotonically increasing cost function $f(x)$, say
>
> $$f(x) = x^3$$
>
> to the number of unused (trailing) spaces x on each line (excluding the last one) is minimized.

Notice that, as is common in reusable software, we have formulated the problem in a minimalist form to make it more widely applicable by eliminating specific details about intervening spaces. A similar minimization problem that includes an intervening space character between each word on a line can be reduced to this basic problem simply by extending, by one, the line length, L, as well as the length of each word, $|w_i|$. In fact, by choosing not to hard-code the space into the basic problem, we can also use the same solution to solve the problem for proportionally spaced fonts simply by representing the line length, the space character, and the individual characters in each word by their corresponding numbers of *points*.[53]

An instance of this basic partitioning problem is illustrated in Figure 0-39a. What makes the optimization problem "hard" is the nonlinear global cost function (Figure 0-39b). If all we needed to do was minimize the total amount of unused space at the end of all lines, we could use a *greedy algorithm* to pack in as many words as possible on each line (in linear time) in a single pass (Figure 0-39c). But, because the cost function is more general, we are instead forced to consider every viable partition to see whether it yields a best result.[54]

In this example, it turns out that by choosing not to place the third word on the first line — even though it fits — the resulting sum of the cubes of the number of trailing spaces on each line is reduced by 730 – 250 = 480 (Figure 0-39d).

---

[53] Recall the open-closed principle proposed in section 0.5, which states that the component is open to extension (by clients) but closed to modification (by library developers).

[54] Note that this problem is inherently different from seemingly similar problems (e.g., "bin-packing" or "one-dimensional knapsack") because in this problem the words must be "packed" in the order in which they are provided.

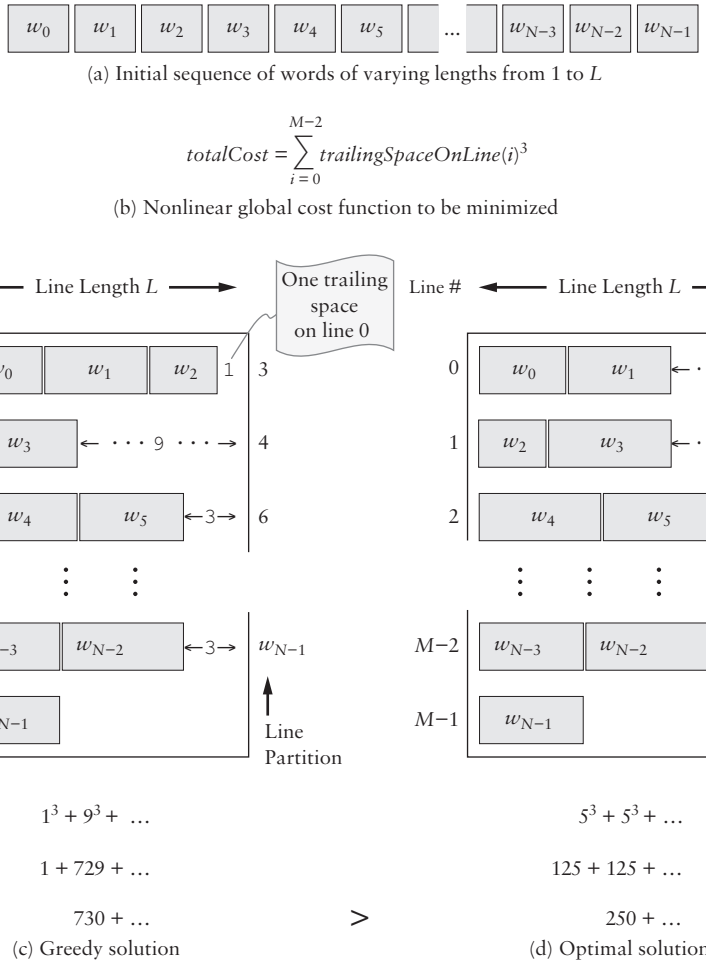


Figure 0-39: Filling text so as to minimize a nonlinear global cost function

Let us take a look at how we might go about solving this problem in software as shown in Figure 0-40. Consistent with our library-based application development methodology, the software solution to this problem — i.e., a program — is implemented as two distinct parts: (1) a single new component having a stable interface and contract that solves the basic problem (Figure 0-40a), and (2) a corresponding driver file containing `main` that adapts this well-factored, stable subsolution so that it can be invoked conveniently on arbitrary text from the

command line (Figure 0-40b). All that remains now is to provide an effective implementation of `FormatUtil::calculateOptimalPartition` in the library component's `.cpp` file.

```
// xyza_formatutil.h

// ...                    (ignoring include guards for now)          (See section 1.5.)

#include <vector>

// ...                    (ignoring include guards for now)          (See section 2.4.)

struct FormatUtil {
    // Provide a namespace[55] for a suite of functions used in formatting text.

    // TYPES
    typedef double (*CostFunction)(int);[56]
        // Alias for a pointer to a C-style function taking a single integer
        // argument (i.e., the number of trailing spaces) and returning a
        // (non-negative) 'double' value.

    // CLASS METHODS
    static void calculateOptimalPartition(
                                        std::vector<int>        *result,
                                        const std::vector<int>&  wordLengths,
                                        int                      lineLength,
                                        CostFunction             costFunction);
        // Load, into the specified 'result', the sequence of respective
        // word indices for the specified 'wordLengths' and 'lineLength'
        // at which to insert line breaks (before the word) in order to
        // minimize the cumulative result of applying the specified
        // 'costFunction' to the number of remaining spaces at the end
        // of each line except the last one.  The behavior is undefined
        // unless, for each value of 'i', '1 <= wordLengths[i] <= lineLength'
        // and the cost function is a monotonically (at least linearly)
        // increasing, non-negative-valued function of its argument.  Note
        // that this primitive calculation packs words with no intervening
        // space; adding 1 to each element of 'wordLengths' as well as to
        // the 'lineLength' achieves the most frequently desired result.

    // ...
};

// ...
```

(a) Stable component header for core text partitioning function

[55] Yes, we *could* have made `FormatUtil` a `namespace` instead of a `struct`, but we deliberately chose not to do so. (See Figure 2-23, section 2.4.9.)

[56] Note that in C++11 and later, instead of a function pointer, we might consider passing either an `std::function` or making the method a function template so as to allow the use of *lambdas* (function literals).

```
// myapplication.cpp
// ...

#include <xyza_formatutil.h>

#include <vector>
#include <iostream>
#include <cctype>   // for 'isspace'
#include <cstddef>  // for 'atoi'
#include <cassert>                                          (See Volume II, section 6.8.)

// ...

double calculateLineCost(int numTrailingSpaces)
    // Return the cost of a line as a pseudo piece-wise continuous
    // (non-sublinear) function of the specified 'numTrailingSpaces'.
    // The behavior is undefined unless '0 <= 'numTrailingSpaces'.
{
    assert(0 <= numTrailingSpaces);                         (See Volume II, section 6.8.)
    double t = numTrailingSpaces;
    return t * t * t;
}

int loadWord(std::string *result)[57]
    // Load into the specified 'result' the next word from standard input.
    // Return 0 on success, and a non-zero value otherwise.  Note that,
    // for the purposes of this example application, a word is a set of
    // contiguous printable (i.e., non-whitespace) characters.

{
    assert(result);                                     (See Volume II, section 6.8.[58])

    // First, skip over whitespace.

    char c = ' ';
    while (std::cin && isspace(c)) {
        std::cin.get(c);
    }

    if (!std::cin) {
        return -1;
    }

    // Found the beginning of a word.  Load the characters from the input
    // stream to 'result'.
```

(continues)

[57] We could have used `<iostream>` directly from the `main` program; instead, we inserted this subroutine, in part, to introduce our coding standards. In our methodology — especially at the library level — we ensure that our code is *exception safe* in that it correctly propagates injected exceptions via RAII (Resource Acquisition Is Initialization), but generally is *exception agnostic* in that we do not `try`, `catch`, or `throw` exceptions directly (see Volume II, section 6.1), preferring instead to return error status. Note also that to signal that an argument to a function (1) is modifiable, (2) its address is retained beyond the call of the function, or (3) is optional, we pass its address; if none of these unusual conditions applies, we pass the argument by value if it is a fundamental, enumerated, or pointer type, and by `const` reference otherwise (see Volume II, section 6.12).

(continued)

```
    result->clear();

    while (std::cin && !isspace(c)) {
        result->push_back(c);
        std::cin.get(c);
    }

    return 0;
}

int main(int argc, const char *argv[])
{
    // Get argument(s), specifically the line length, which defaults to 79.

    const int lineLength = argc > 1 ? std::atoi(argv[1]) : 79;

    // Read in words.  Note that we add one to each word length to include a
    // trailing space.

    std::vector<std::string> words;
    std::vector<int>         wordLengths;

    std::string word;
    while (0 == loadWord(&word)) {
        words.push_back(word);
        wordLengths.push_back(word.length() + 1);
    }

    // Process word lengths.

    std::vector<int> partition;

    FormatUtil::calculateOptimalPartition(   // ignoring namespaces for now
        &partition,
        wordLengths,
        lineLength + 1,   // Adjusted for trailing space after last word.
        calculateLineCost
    );

    // Format output, placing all words within a single partition on one line.
    // Note that, by pushing the total number of words onto the end of the
    // partition list, we are converting a list of separators into a list of
```

(continues)

[58] A preliminary version of a (language-based) contract-checking facility fashioned after the library-based solution advocated in section 6.8 of Volume II of this book was adopted (temporarily) into a draft of the C++20 Standard in June 2018 but was withdrawn a year later to allow time for further consideration and refinement.

(continued)

```
    // terminators, where each partition indicates the word after which to
    // terminate the line.

    int i = 0;
    int partitionIndex = 0;
    partition.push_back(words.size());

    while (partitionIndex < partition.size()) {
        while (i < partition[partitionIndex] - 1) {
            std::cout << words[i++] << ' ';
        }
        std::cout << words[i++] << std::endl;
        ++partitionIndex;
    }
}
```

(b) Malleable top-level text-partitioning driver file containing `main`

Figure 0-40: Component-based decomposition of text-partitioning program

A first attempt at a solution to the basic problem addressed by the library component might be to iterate over all of the potential partition (i.e., newline) indices $\{1 \ldots N - 1\}$ in turn, solve each pair of subproblems recursively, and then select a solution pair having the lowest combined cost as the result. A pseudocode implementation of this algorithm is provided in Figure 0-41a; the actual C++ implementation provided in Figure 0-41b serves to make this illustration concrete.

```
If [A .. B] fits on a line

    If this is the last line (i.e., B is last word)
        Return 0.0.
    Else
        Return cost(number of trailing spaces on this line).

Else // [A .. B] doesn't fit

    For each division position [A + 1 .. B]
    {
        • Divide the text into left and right subproblems.
        • Separately solve each left/right subproblem recursively.
        • Record index and subpartitions of pair with min combined cost.
    }

Load into result: [ left-partition, index, right-partition ].

Return the combined costs of the left and right partitions.
```

(a) Pseudocode

```
// xyza_formatutil.cpp
// ...

#include <xyza_formatutil.h>
// ...

namespace {
                        // -------------
                        // LOCAL CONTEXT
                        // -------------

struct Context {
    // This structure holds "global" information used during the recursion.

    // DATA
    const std::vector<int>&  d_wordLengths;   // length of each word in text
    int                      d_lineLength;    // maximum length of each line
    FormatUtil::CostFunction d_costFunction;  // applied to trailing space

    // CREATORS
    Context(const std::vector<int>&  wordLengths,
            int                      lineLength,
            FormatUtil::CostFunction costFunction)
    : d_wordLengths(wordLengths)
    , d_lineLength(lineLength)
    , d_costFunction(costFunction)
    {
    }
};

}  // close unnamed namespace

                        // --------------------
                        // RECURSIVE SUBROUTINE
                        // --------------------

static double minCost1(std::vector<int> *result,
                       int               a,
                       int               b,
                       const Context&    context)
    // Load, into the specified 'result', an optimal subsequence of
    // respective word indices in the specified (inclusive) range '[a, b]'
    // at which to insert line breaks as dictated by the specified 'context'.
    // Return the total cost of the optimal solution.  The behavior is
    // undefined unless '0 <= a', 'a <= b', and
    // 'b < context.d_wordLengths.size()'.
{
    assert(result);                                          (See Volume II, section 6.8.)
    assert(0 <= a);                                          (See Volume II, section 6.8.)
    assert(a <= b);                                          (See Volume II, section 6.8.)
    assert(b < context.d_wordLengths.size());                (See Volume II, section 6.8.)
    double resultCost;  // value to be returned (set below)
    result->clear();    // Make sure that result is initially empty.

    // First see if current region [a, b] will fit on a line.

    int sum = 0;
```

(continues)

(continued)

```
    int i;
    for (i = a; i <= b && sum <= context.d_lineLength; ++i) {
        sum += context.d_wordLengths[i];
    }

    if (i > b && sum <= context.d_lineLength) { // if fits?
        resultCost = context.d_wordLengths.size() - 1 == b    // if last line
                     ? 0.0                                    // then no charge
                     : context.d_costFunction(context.d_lineLength - sum);
                                                              // else get cost
    }
    else if (a == b) {          // Although the behavior is unspecified if a
        resultCost = 0.0;       // single word is too long, the only reasonable
    }                           // thing to do is pass it back at no charge.
    else {
        assert(a < b);

        // The current sequence is too long and must be further partitioned.
        // For each possible partition location 'k', 'a < k <= b', solve both
        // the left- and right-side problems recursively.  If the combined
        // cost of the two subpartitions is less than any so far, record 'k'
        // and the combined cost.  Finally, append 'left', 'k', and 'right' to
        // the result array and return the minimum cost.

        double lowestCost;
        int    lowestK;

        std::vector<int> lowestLeft;
        std::vector<int> lowestRight;
        std::vector<int> left;
        std::vector<int> right;

        const int first = a + 1;

        for (int k = first; k <= b; ++k) {
            double cost = minCost1(&left,  a, k - 1, context)
                        + minCost1(&right, k, b,     context);

            if (first == k || cost < lowestCost) {
                lowestCost  = cost;
                lowestK     = k;
                lowestLeft  = left;
                lowestRight = right;
            }
        }

        result->insert(result->end(),lowestLeft.begin(), lowestLeft.end());
        result->push_back(lowestK);
        result->insert(result->end(),lowestRight.begin(), lowestRight.end());
        resultCost = lowestCost;
    }
```

Active comment

(continues)

(continued)

```
    // '*result' holds an optimal sub-solution.

    return resultCost;
}

                        // -----------------
                        // TOP-LEVEL ROUTINE
                        // -----------------

// CLASS METHODS
void FormatUtil::calculateOptimalPartition(
                                std::vector<int>         *result,
                                const std::vector<int>&   wordLengths,
                                int                       lineLength,
                                FormatUtil::CostFunction  costFunction)
{
    assert(result);                                          (See Volume II, section 6.8.)
    assert(0 <= lineLength);                                 (See Volume II, section 6.8.)

    // Solve entire problem recursively.

    Context context(wordLengths, lineLength, costFunction);

    minCost1(result,                  // where to load partition
             0,                       // a
             wordLengths.size() - 1,  // b
             context);                // global info
}

// ...
```

(b) Actual C++ implementation

Figure 0-41: Brute-force, recursive top-down partitioning

In many important ways, a brute-force recursive solution to this relatively simple text partitioning problem mirrors the kinds of gross inefficiencies that routinely afflict large-scale software development: An enormous amount of time and effort is squandered partitioning an initial problem into subproblems and then repeatedly repartitioning those problems into sub-subproblems, and so on, until a problem of manageable size is reached. In the case of text partitioning, "manageable size" means it all fits on a single line, whereas in software our unit of manageable size is the component.

To see how this text-partitioning problem maps almost directly onto the top-down decomposition of application software, let us examine an instance of the computation in detail. The `Context` in Figure 0-41b (analogous to the software development environment itself) comprises all the immutable parameters (analogous to compilers, debuggers, third-party libraries, etc.) needed to perform the recursive calculation efficiently. The top-level routine, `FormatUtil::calculateOptimalPartition`, creates an instance of the `Context`

structure on the program stack (achieving thread safety) and then invokes the recursive subroutine `minCost1`.

The first recursion step of this brute-force text-partitioning algorithm (analogous to decomposing an application into subsystems)[59] leads to a fan-out of $N - 1$ pairs of subroutine calls. Each pair, in turn, divides the initial problem into unique left/right subproblems (analogous to potential subsystem designs). Each subproblem is solved recursively, again dividing the subproblem into all possible left/right sub-subproblem pairs (sub-subsystems), and so on, to a worst-case depth of N. In this analogy, the words (or, more precisely, word lengths) correspond to logical content (i.e., behavior/functionality) to be partitioned (i.e., designed) into physical lines (i.e., components).

Just as in top-down software design, the recursion terminates when the subregion is sufficiently small so as to fit on a line (or component) or when the subregion consists of only a single word (or the logical content of the component is atomic). It is only at this point that a cost can be assessed (i.e., the leaf component developed) and returned (i.e., released to production).

For the leaf case, the subpartition (returned via the argument list)[60] is empty. For every other case, the resulting subpartition will be determined by whichever left-right pair of subregions yields the lowest combined cost (best design). The task of determining an optimal subregion pair (implementing a subsystem) requires evaluating and comparing the results of each of the subpartitions (subsystem designs). In the end, an optimal partition (the integrated subsystem) is returned along with its total cost to the caller (the business sponsor).

Figure 0-42 illustrates the brute-force algorithm of Figure 0-41 (analogous to pure top-down design) applied to $N = 5$ words, each of which (just for computational simplicity) we'll assume here to be of length equal to the line length ($L = 1$). At every step in the recursion, each of the $2 \cdot (M - 1)$ subpartitions of the subsequence of length $M \leq N$ is locally unique. For example, [bcde] decomposes into the three left-right subsequence pairs [b:cde], [bc:de], and [bcd:e], with each of the six subsequences — e.g., [bc] — indicated by a different (i, j) pair of word

[59] We acknowledge that the task of partitioning application software is qualitatively harder than our unidimensional text partitioning analogy. Still, the concreteness of this example serves to illustrate quantitatively what are essentially the same points.

[60] The resulting partition is represented here as a sequence of word indices that are loaded (i.e., values copied) into the modifiable `std::vector<int>` whose address was passed in as the first argument (see Volume II, section 6.12) to the recursive function. Even in modern C++ (i.e., C++11 and later), returning — by value — objects that allocate dynamic memory in what is intended to be hierarchically reusable software infrastructure remains in our view misguided — e.g., because it explicitly prevents object pooling across invocations (see Volume II, section 4.10).

indices — e.g., (1, 2). Hence, every time a partitioning problem is encountered recursively, it is resolved anew leading to an exponential amount of work — in this case, 3^{N-1} subproblems.[61]

Theoretically, this simple algorithm is correct and will eventually produce an optimal solution; its exponential run time, however, makes it infeasible for problems of any substantial size *N*. Just as with pure top-down application development, this naive approach to partitioning does not scale.

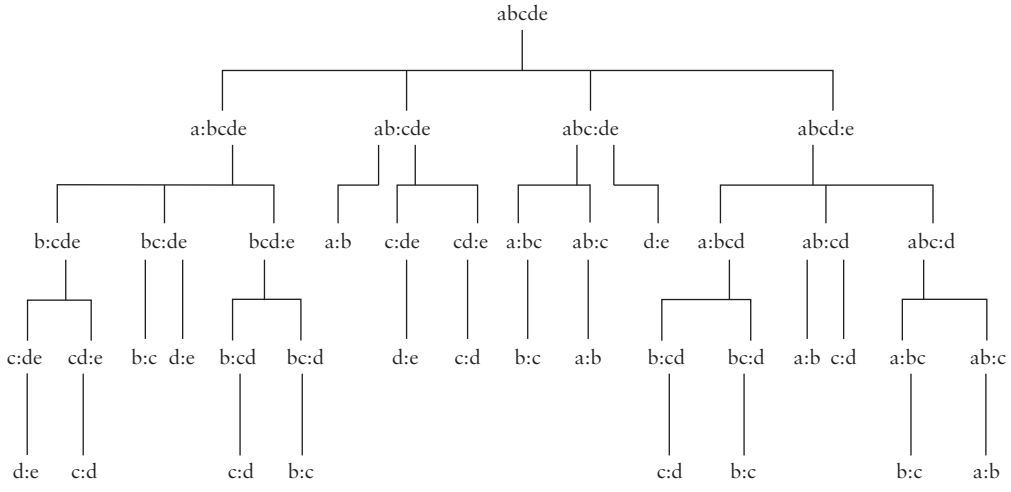


Figure 0-42: Exponential effort for naive partitioning (NO REUSE)

---

[61] This is a demonstration of recursion being 3^{N-1} for the simplified problem where `lineLength` = `wordLength`:

$$f(N) = \sum_{i=1}^{N-1} f(i) + f(N-i)$$

$$f(N) = 2\sum_{i=1}^{N-1} f(i)$$

$$f(N) = 2(\sum_{i=1}^{N-2} f(i)) + 2f(N-1)$$

$$f(N) = 3f(N-1)$$

$$f(N) = 3^{N-1}$$

This analysis, however, counts only the number of calls to *f* and does not address the runtime cost. For run time, we need to add overhead constants:

$$f(N) = (\sum_{i=1}^{N-2} (f(i) + f(N-i) + C_1)) + C_2$$

The key observation here is that we are doing far more work than is necessary. If there were truly an exponential number of distinct, stable subproblems, we might reasonably expect to spend an exponential amount of time solving them. Yet consider that we were able to represent each unique subproblem with just an integer pair (i, j), where $0 \le i \le j < N$, which means that the total number of unique subproblems is not exponential in N but merely quadratic.

$$\textit{Number of unique subproblems} = \frac{N(N+1)}{2}$$

Just as in top-down application design, the number of unique partitions (like useful, well-factored software subsystems) is far smaller than the number of effective *uses* of them. If we could solve each subproblem just *once*, we'd be done in polynomial time!

To avoid the exponential costs associated with top-down design, we need to find a way to avoid solving the same subproblem over and over again. Suppose that each time we calculate an optimal subpartition we cache that solution and make it globally available. Suppose further that, instead of blindly recalculating each subproblem recursively, we first make an attempt to look up the solution in the global cache in case the very problem we are working on has already been solved.

This classic optimization technique, known as *dynamic programming*[62] (a.k.a. *memoization*), has been used successfully by many, including this author,[63] for reducing seemingly exponentially hard problems to tractable ones that can be solved in polynomial time. By simply recording and making available the optimal solution for each of the relatively few unique partitions (or subsystems) we create, the amount of time and effort expended to solve this simple text partitioning problem (or application) is reduced dramatically. What is more, unlike text partitioning where hierarchical reuse is specific to a single instance of a problem (e.g., Figure 0-11a, section 0.3), the analogous solution cache in software — i.e., enterprise-wide hierarchically reusable software libraries (see section 0.9) — can span multiple versions of multiple software applications and products (e.g., Figure 0-11c, section 0.3).

Figure 0-43 illustrates, in pseudocode, what would be necessary to achieve the proposed optimization. Notice that the previous pseudocode is unchanged (i.e., stable). Instead, two new blocks of code are added: one at the beginning to determine whether the solution to this subproblem is already known and one at the end to record each new solution for future use. The obvious analogy is that software engineers are well-advised to look for an appropriate existing solution — or at

---

[62] **bellman54**

[63] For example, see **lakos97a**.

least appropriate subparts — before creating an entirely new one from scratch. If creating a new solution is necessary, it might ultimately be appropriate to render and package it in a manner that allows it to be discovered and reused to address similar needs in the future (see section 0.10).

```
If [A .. B] is found in solution map
   Copy partition into result and return cost.

If [A .. B] fits on a line

   If this is the last line (i.e., B is last word)
      Return 0.0.
   Else
      Return cost(number of trailing spaces on this line).

Else // [A .. B] doesn't fit

   For each division position [A + 1 .. B]
   {
     • Divide the text into left and right subproblems.
     • Separately solve each left/right subproblem recursively.
     • Record index and subpartitions of pair with min combined cost.
   }

Load into result: [ left-partition, index, right-partition ].

Record cost and partition for [A .. B] in solution map.

Return the combined costs of the left and right partitions.
```

Figure 0-43: Pseudocode employing dynamic programming

Figure 0-44a shows how we would modify the `Context struct` of Figure 0-39b to contain a mutable cache, `d_solutionCache`, implemented as a mapping from a pair of integers, uniquely identifying the subrange of word indices, to a `Solution struct` containing (1) an optimal sequence of word indices at which to introduce line breaks, and (2) the cost associated with trailing spaces for that partition. When the `Context` object is created, `d_solutionCache` is initially empty; however, each time the modified `minCost1` subroutine (Figure 0-44b) calculates a solution for a new subrange `[`*i, j*`]`, that solution is retained so that it will be readily available the next time it is needed. Again, notice that this optimization introduced some new source code but didn't require changes to *any* existing code.

```
// xyza_formatutil.cpp
// ...

#include <xyza_formatutil.h>
// ...

namespace {
```

NEW

```
                        // --------------
                        // SOLUTION CACHE
                        // --------------

typedef std::pair<int, int> Range;
    // Alias for a pair of integers representing a range of word indices.

struct Solution {
    // This structure holds the solution/cost for a particular sub-range.

    std::vector<int> d_partition;  // word indices to receive line breaks
    double           d_cost;       // cost associated with this solution
};

typedef std::map<Range, Solution> Map;
    // Alias for a mapping of index ranges to optimal partitions/total costs.
```

```
                        // -------------
                        // LOCAL CONTEXT
                        // -------------
struct Context {
    // This structure holds "global" information used during the recursion.

    // DATA
    const std::vector<int>&  d_wordLengths;   // length of each word in text
    int                      d_lineLength;    // maximum length of each line
    FormatUtil::CostFunction d_costFunction;  // applied to trailing space
```

NEW

```
    mutable Map              d_solutionCache; // ranges/solution association
```

```
    // CREATORS
    Context(const std::vector<int>&  wordLengths,
            int                      lineLength,
            FormatUtil::CostFunction costFunction)
    : d_wordLengths(wordLengths)
    , d_lineLength(lineLength)
    , d_costFunction(costFunction)
    { }
};

}  // close unnamed namespace

// ...
```

(a) Revised `Context` `struct` with mutable solution cache