Large-Scale C++

Volume I

Process and Architecture

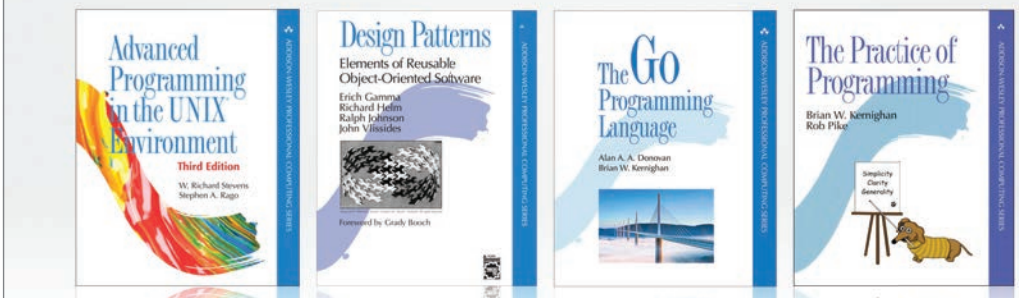
John Lakos

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]

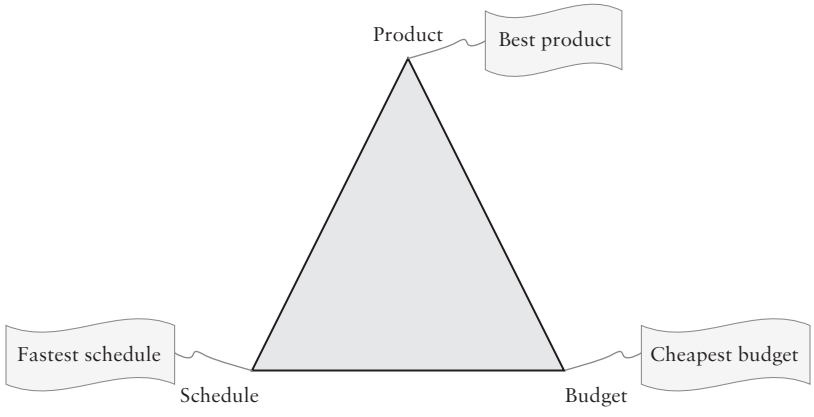


Figure 0-1: Schedule/product/budget trade-offs

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.

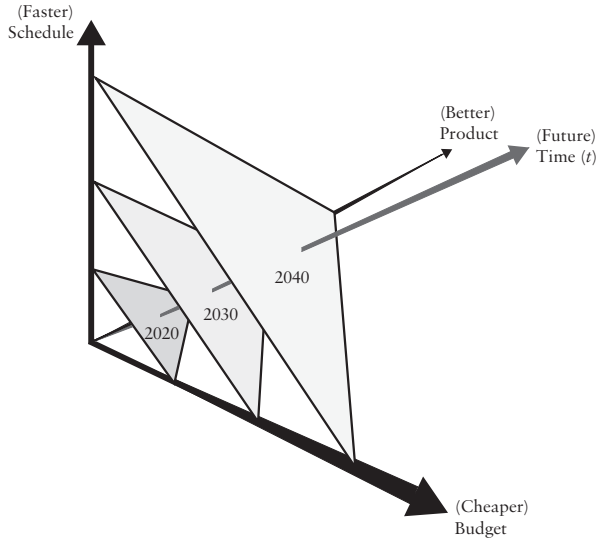


Figure 0-2: Improving the schedule/product/budget design space

---

[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.

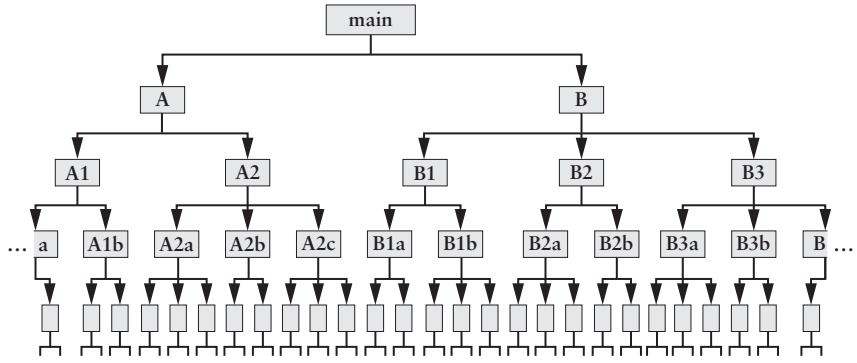


Figure 0-3: Pure top-down design (BAD IDEA)

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]

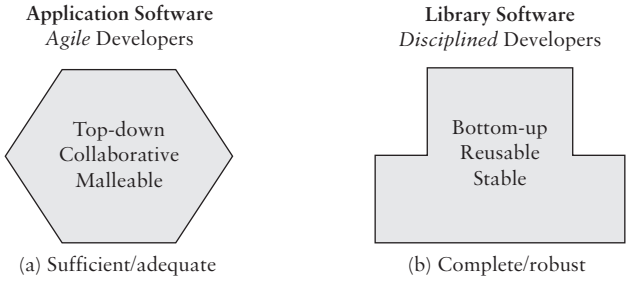


Figure 0-4: Library versus application software development

[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