Large-Scale C++
Volume I
Process and Architecture
John Lakos
Large-Scale C++
This page intentionally left blank
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 xvii
Acknowledgments xxv

Chapter 0: Motivation 1
0.1 The Goal: Faster, Better, Cheaper! ................................................................. 3
0.2 Application vs. Library Software ................................................................. 5
0.3 Collaborative vs. Reusable Software .......................................................... 14
0.4 Hierarchically Reusable Software ............................................................... 20
0.5 Malleable vs. Stable Software .............................................................. 29
0.6 The Key Role of Physical Design .............................................................. 44
0.7 Physically Uniform Software: The Component ....................................... 46
0.8 Quantifying Hierarchical Reuse: An Analogy ........................................... 57
0.9 Software Capital ......................................................................................... 86
0.10 Growing the Investment .......................................................................... 98
0.11 The Need for Vigilance ........................................................................... 110
0.12 Summary ................................................................................................. 114

Chapter 1: Compilers, Linkers, and Components 123
1.1 Knowledge Is Power: The Devil Is in the Details ........................................ 125
1.1.1 “Hello World!” ....................................................................................... 125
1.1.2 Creating C++ Programs ......................................................................... 126
1.1.3 The Role of Header Files ....................................................................... 128
1.2 Compiling and Linking C++ .................................................................... 129
1.2.1 The Build Process: Using Compilers and Linkers .............................. 129
1.2.2 Classical Atomicity of Object (.o) Files ............................................... 134
1.2.3 Sections and Weak Symbols in .o Files ................................................................. 138
1.2.4 Library Archives ................................................................................................. 139
1.2.5 The “Singleton” Registry Example ..................................................................... 141
1.2.6 Library Dependencies ....................................................................................... 146
1.2.7 Link Order and Build-Time Behavior ............................................................... 151
1.2.8 Link Order and Runtime Behavior .................................................................... 152
1.2.9 Shared (Dynamically Linked) Libraries ............................................................ 153
1.3 Declarations, Definitions, and Linkage ............................................................... 153
1.3.1 Declaration vs. Definition ................................................................................ 154
1.3.2 (Logical) Linkage vs. (Physical) Linking ......................................................... 159
1.3.3 The Need for Understanding Linking Tools ...................................................... 160
1.3.4 Alternate Definition of Physical “Linkage”: Bindage ....................................... 160
1.3.5 More on How Linkers Work ............................................................................ 162
1.3.6 A Tour of Entities Requiring Program-Wide Unique Addresses ..................... 163
1.3.7 Constructs Where the Caller’s Compiler Needs the Definition’s Source Code ...... 166
1.3.8 Not All Declarations Require a Definition to Be Useful .................................... 168
1.3.9 The Client’s Compiler Typically Needs to See Class Definitions ................. 169
1.3.10 Other Entities Where Users’ Compilers Must See the Definition ................. 170
1.3.11 Enumerations Have External Linkage, but So What?! ................................. 170
1.3.12 Inline Functions Are a Somewhat Special Case .......................................... 171
1.3.13 Function and Class Templates ..................................................................... 172
1.3.14 Function Templates and Explicit Specializations ......................................... 172
1.3.15 Class Templates and Their Partial Specializations ..................................... 179
1.3.16 extern Templates ......................................................................................... 183
1.3.17 Understanding the ODR (and Bindage) in Terms of Tools ............................ 185
1.3.18 Namespaces .................................................................................................. 186
1.3.19 Explanation of the Default Linkage of const Entities ................................. 188
1.3.20 Summary of Declarations, Definitions, Linkage, and Bindage ................. 188
1.4 Header Files ........................................................................................................ 190
1.5 Include Directives and Include Guards ............................................................. 201
1.5.1 Include Directives ......................................................................................... 201
1.5.2 Internal Include Guards ............................................................................... 203
1.5.3 (Deprecated) External Include Guards ......................................................... 205
1.6 From .h /.cpp Pairs to Components .................................................................. 209
1.6.1 Component Property 1 ................................................................................ 210
1.6.2 Component Property 2 ................................................................................ 212
1.6.3 Component Property 3 ................................................................................ 214
1.7 Notation and Terminology ................................................................................. 216
1.7.1 Overview ...................................................................................................... 217
1.7.2 The Is-A Logical Relationship ..................................................................... 219
1.7.3 The Uses-In-The-Interface Logical Relationship .......................................... 219
1.7.4 The Uses-In-The-Implementation Logical Relationship ............................... 221
1.7.5 The Uses-In-The-Interface Logical Relationship and the Protocol Class .... 226
1.7.6 In-Structure-Only (ISO) Collaborative Logical Relationships .................... 227
1.7.7 How Constrained Templates and Interface Inheritance Are Similar ............ 230
1.7.8  How Constrained Templates and Interface Inheritance Differ ...........................................232
1.7.8.1 Constrained Templates, but Not Interface Inheritance ........................................232
1.7.8.2 Interface Inheritance, but Not Constrained Templates ........................................233
1.7.9 All Three “Inheriting” Relationships Add Unique Value ...................................................234
1.7.10 Documenting Type Constraints for Templates ............................................................234
1.7.11 Summary of Notation and Terminology .............................................................................237
1.8 The Depends-On Relation..................................................................................................237
1.9 Implied Dependency .......................................................................................................243
1.10 Level Numbers ...............................................................................................................251
1.11 Extracting Actual Dependencies ..........................................................................................256
1.11.1 Component Property 4 ..................................................................................................257
1.12 Summary ..................................................................................................................259

Chapter 2: Packaging and Design Rules 269
2.1 The Big Picture ........................................................................................................270
2.2 Physical Aggregation ....................................................................................................275
2.2.1 General Definition of Physical Aggregate ...............................................................275
2.2.2 Small End of Physical-Aggregation Spectrum .........................................................275
2.2.3 Large End of Physical-Aggregation Spectrum ............................................................277
2.2.4 Conceptual Atomicity of Aggregates ..............................................................................277
2.2.5 Generalized Definition of Dependencies for Aggregates .................................................278
2.2.6 Architectural Significance .............................................................................................278
2.2.7 Architectural Significance for General UORs ..............................................................279
2.2.8 Parts of a UOR That Are Architecturally Significant ..................................................279
2.2.9 What Parts of a UOR Are Not Architecturally Significant? .........................................279
2.2.10 A Component Is “Naturally” Architecturally Significant ..............................................280
2.2.11 Does a Component Really Have to Be a .h/.cpp Pair? ................................................280
2.2.12 When, If Ever, Is a .h/.cpp Pair Not Good Enough? ..................................................280
2.2.13 Partitioning a .cpp File Is an Organizational-Only Change .......................................281
2.2.14 Entity Manifest and Allowed Dependencies ................................................................281
2.2.15 Need for Expressing Envelope of Allowed Dependencies ........................................284
2.2.16 Need for Balance in Physical Hierarchy ..............................................................284
2.2.17 Not Just Hierarchy, but Also Balance ..........................................................................285
2.2.18 Having More Than Three Levels of Physical Aggregation Is Too Many .......................287
2.2.19 Three Levels Are Enough Even for Larger Systems ....................................................289
2.2.20 UORs Always Have Two or Three Levels of Physical Aggregation ............................289
2.2.21 Three Balanced Levels of Aggregation Are Sufficient. Trust Me! .................................290
2.2.22 There Should Be Nothing Architecturally Significant Larger Than a UOR .................290
2.2.23 Architecturally Significant Names Must Be Unique ....................................................292
2.2.24 No Cyclic Physical Dependencies! ...............................................................................293
2.2.25 Section Summary .......................................................................................................293
2.3 Logical/Physical Coherence ............................................................................................294
2.14 The Hierarchical Testability Requirement .........................................................................................................................437
  2.14.1 Leveraging Our Methodology for Fine-Grained Unit Testing .........................................................................................438
  2.14.2 Plan for This Section (Plus Plug for Volume II and Especially Volume III) .................................................................438
  2.14.3 Testing Hierarchically Needs to Be Possible .........................................................................................................................439
  2.14.4 Relative Import of Local Component Dependencies with Respect to Testing .................................................................447
  2.14.5 Allowed Test-Driver Dependencies Across Packages .........................................................................................................451
  2.14.6 Minimize Test-Driver Dependencies on the External Environment ..................................................................................454
  2.14.7 Insist on a Uniform (Standalone) Test-Driver Invocation Interface ..................................................................................456
  2.14.8 Section Summary .................................................................................................................................................................458

2.15 From Development to Deployment .................................................................................................................................459
  2.15.1 The Flexible Deployment of Software Should Not Be Compromised ..................................................................................459
  2.15.2 Having Unique .h and .o Names Are Key ...............................................................................................................................460
  2.15.3 Software Organization Will Vary During Development ...................................................................................................460
  2.15.4 Enterprise-Wide Unique Names Facilitate Refactoring .....................................................................................................461
  2.15.5 Software Organization May Vary During Just the Build Process ....................................................................................462
  2.15.6 Flexibility in Deployment Is Needed Even Under Normal Circumstances ...........................................................................462
  2.15.7 Flexibility Is Also Important to Make Custom Deployments Possible ..................................................................................462
  2.15.8 Flexibility in Stylistic Rendering Within Header Files ........................................................................................................463
  2.15.9 How Libraries Are Deployed Is Never Architecturally Significant ....................................................................................464
  2.15.10 Partitioning Deployed Software for Engineering Reasons .................................................................................................464
  2.15.11 Partitioning Deployed Software for Business Reasons ......................................................................................................467
  2.15.12 Section Summary .................................................................................................................................................................469

2.16 Metadata .................................................................................................................................................................................469
  2.16.1 Metadata Is “By Decree” .......................................................................................................................................................470
  2.16.2 Types of Metadata .................................................................................................................................................................471
    2.16.2.1 Dependency Metadata ......................................................................................................................................................471
    2.16.2.2 Build Requirements Metadata ........................................................................................................................................475
    2.16.2.3 Membership Metadata ......................................................................................................................................................476
    2.16.2.4 Enterprise-Specific Policy Metadata ........................................................................................................................................476
  2.16.3 Metadata Rendering .................................................................................................................................................................478
  2.16.4 Metadata Summary .................................................................................................................................................................479

2.17 Summary ..................................................................................................................................................................................481

Chapter 3: Physical Design and Factoring ....................................................................................................................................495

3.1 Thinking Physically .................................................................................................................................................................497
  3.1.1 Pure Classical (Logical) Software Design Is Naive ...................................................................................................................497
  3.1.2 Components Serve as Our Fine-Grained Modules ..................................................................................................................498
  3.1.3 The Software Design Space Has Direction ...............................................................................................................................498
    3.1.3.1 Example of Relative Physical Position: Abstract Interfaces .................................................................................................498
  3.1.4 Software Has Absolute Location ...............................................................................................................................................500
    3.1.4.1 Asking the Right Questions Helps Us Determine Optimal Location ....................................................................................500
    3.1.4.2 See What Exists to Avoid Reinventing the Wheel ...................................................................................................................500
    3.1.4.3 Good Citizenship: Identifying Proper Physical Location ........................................................................................................501
3.1.5 The Criteria for Colocation Should Be Substantial, Not Superficial
3.1.6 Discovery of Nonprimitive Functionality Absent Regularity Is Problematic
3.1.7 Package Scope Is an Important Design Consideration
  3.1.7.1 Package Charter Must Be Delineated in Package-Level Documentation
  3.1.7.2 Package Prefixes Are at Best Mnemonic Tags, Not Descriptive Names
  3.1.7.3 Package Prefixes Force Us to Consider Design More Globally Early
  3.1.7.4 Package Prefixes Force Us to Consider Package Dependencies
  from the Start
  3.1.7.5 Even Opaque Package Prefixes Grow to Take On Important Meaning
  3.1.7.6 Effective (e.g., Associative) Use of Package Names Within Groups
3.1.8 Limitations Due to Prohibition on Cyclic Physical Dependencies
3.1.9 Constraints on Friendship Intentionally Preclude Some Logical Designs
3.1.10 Introducing an Example That Justifiably Requires Wrapping
  3.1.10.1 Wrapping Just the Time Series and Its Iterator in a Single Component
  3.1.10.2 Private Access Within a Single Component Is an Implementation Detail
  3.1.10.3 An Iterator Helps to Realize the Open-Closed Principle
  3.1.10.4 Private Access Within a Wrapper Component Is Typically Essential
  3.1.10.5 Since This Is Just a Single-Component Wrapper, We Have Several Options
  3.1.10.6 Multicomponent Wrappers, Not Having Private Access, Are Problematic
  3.1.10.7 Example Why Multicomponent Wrappers Typically Need “Special” Access
  3.1.10.8 Wrapping Interoperating Components Separately Generally Doesn’t Work
  3.1.10.9 What Should We Do When Faced with a Multicomponent Wrapper?
3.1.11 Section Summary

3.2 Avoiding Poor Physical Modularity
  3.2.1 There Are Many Poor Modularization Criteria; Syntax Is One of Them
  3.2.2 Factoring Out Generally Useful Software into Libraries Is Critical
  3.2.3 Failing to Maintain Application/Library Modularity Due to Pressure
  3.2.4 Continuous Demotion of Reusable Components Is Essential
    3.2.4.1 Otherwise, in Time, Our Software Might Devolve into a
           “Big Ball of Mud”!
  3.2.5 Physical Dependency Is Not an Implementation Detail to an App Developer
  3.2.6 Iterators Can Help Reduce What Would Otherwise Be Primitive Functionality
  3.2.7 Not Just Minimal, Primitive: The Utility struct
  3.2.8 Concluding Example: An Encapsulating Polygon Interface
    3.2.8.1 What Other UDTs Are Used in the Interface?
    3.2.8.2 What Invariants Should our::Polygon Impose?
    3.2.8.3 What Are the Important Use Cases?
    3.2.8.4 What Are the Specific Requirements?
    3.2.8.5 Which Required Behaviors Are Primitive and Which Aren’t?
    3.2.8.6 Weighing the Implementation Alternatives
    3.2.8.7 Achieving Two Out of Three Ain’t Bad
    3.2.8.8 Primitiveness vs. Flexibility of Implementation
    3.2.8.9 Flexibility of Implementation Extends Primitive Functionality
    3.2.8.10 Primitiveness Is Not a Draconian Requirement
3.2.8.11  What About Familiar Functionality Such as Perimeter and Area? ...................537
3.2.8.12  Providing Iterator Support for Generic Algorithms ........................................539
3.2.8.13  Focus on Generally Useful Primitive Functionality .......................................540
3.2.8.14  Suppress Any Urge to Colocate Nonprimitive Functionality ...........................541
3.2.8.15  Supporting Unusual Functionality .....................................................................541
3.2.9   Semantics vs. Syntax as Modularization Criteria .....................................................552
3.2.9.1  Poor Use of $u$ as a Package Suffix ....................................................................552
3.2.9.2  Good Use of util as a Component Suffix ..........................................................553
3.2.10  Section Summary ..................................................................................................553
3.3 Grouping Things Physically That Belong Together Logically .........................................555
3.3.1   Four Explicit Criteria for Class Colocation .............................................................555
3.3.1.1  First Reason: Friendship .........................................................................................556
3.3.1.2  Second Reason: Cyclic Dependency ..................................................................557
3.3.1.3  Third Reason: Single Solution .............................................................................557
3.3.1.4  Fourth Reason: Flea on an Elephant ....................................................................559
3.3.2   Colocation Beyond Components .............................................................................560
3.3.3   When to Make Helper Classes Private to a Component .............................................561
3.3.4   Colocation of Template Specializations .................................................................564
3.3.5   Use of Subordinate Components .............................................................................564
3.3.6   Colocate Tight Mutual Collaboration within a Single UOR ....................................565
3.3.7   Day-Count Example ...............................................................................................566
3.3.8   Final Example: Single-Threaded Reference-Counted Functors ...............................576
3.3.8.1  Brief Review of Event-Driven Programming .........................................................576
3.3.8.2  Aggregating Components into Packages ............................................................586
3.3.8.3  The Final Result ..................................................................................................589
3.3.9   Section Summary ...................................................................................................591
3.4 Avoiding Cyclic Link-Time Dependencies .........................................................................592
3.5 Levelization Techniques ..................................................................................................602
3.5.1   Classic Levelization .................................................................................................602
3.5.2   Escalation ..................................................................................................................604
3.5.3   Demotion ...................................................................................................................614
3.5.4   Opaque Pointers .......................................................................................................618
3.5.4.1  Manager/Employee Example ................................................................................618
3.5.4.2  Event/EventQueue Example ...............................................................................623
3.5.4.3  Graph/Node/Edge Example .................................................................................625
3.5.5   Dumb Data ...............................................................................................................629
3.5.6   Redundancy ............................................................................................................634
3.5.7   Callbacks ..................................................................................................................639
3.5.7.1  Data Callbacks ......................................................................................................640
3.5.7.2  Function Callbacks ...............................................................................................643
3.5.7.3  Functor Callbacks ...............................................................................................651
3.5.7.4  Protocol Callbacks ...............................................................................................655
3.5.7.5  Concept Callbacks ...............................................................................................664
### Contents

3.5.8 Manager Class ................................................................. 671
3.5.9 Factoring ............................................................................. 674
3.5.10 Escalating Encapsulation .................................................. 677
  3.5.10.1 A More General Solution to Our Graph Subsystem .......... 681
  3.5.10.2 Encapsulating the Use of Implementation Components ....... 683
  3.5.10.3 Single-Component Wrapper .......................................... 685
  3.5.10.4 Overhead Due to Wrapping ........................................... 687
  3.5.10.5 Realizing Multicomponent Wrappers ............................. 687
  3.5.10.6 Applying This New, “Heretical” Technique to Our Graph Example .... 688
  3.5.10.7 Why Use This “Magic” reinterpret_cast Technique? ............ 692
  3.5.10.8 Wrapping a Package-Sized System ................................. 693
  3.5.10.9 Benefits of This Multicomponent-Wrapper Technique .......... 701
  3.5.10.10 Misuse of This Escalating-Encapsulation Technique .......... 702
  3.5.10.11 Simulating a Highly Restricted Form of Package-Wide Friendship .... 702
3.5.11 Section Summary ......................................................... 703

3.6 Avoiding Excessive Link-Time Dependencies .............................. 704
  3.6.1 An Initially Well-Factored Date Class That Degrades Over Time .......... 705
  3.6.2 Adding Business-Day Functionality to a Date Class (BAD IDEA) .......... 715
  3.6.3 Providing a Physically Monolithic Platform Adapter (BAD IDEA) ........ 717
  3.6.4 Section Summary .......................................................... 722

3.7 Lateral vs. Layered Architectures ............................................. 722
  3.7.1 Yet Another Analogy to the Construction Industry ................. 723
  3.7.2 (Classical) Layered Architectures ....................................... 723
  3.7.3 Improving Purely Compositional Designs ............................... 726
  3.7.4 Minimizing Cumulative Component Dependency (CCD) ............ 727
    3.7.4.1 Cumulative Component Dependency (CCD) Defined ............... 729
    3.7.4.2 Cumulative Component Dependency: A Concrete Example ......... 730
  3.7.5 Inheritance-Based Lateral Architectures .................................. 732
  3.7.6 Testing Lateral vs. Layered Architectures .............................. 738
  3.7.7 Section Summary .......................................................... 738

3.8 Avoiding Inappropriate Link-Time Dependencies .......................... 739
  3.8.1 Inappropriate Physical Dependencies .................................... 740
  3.8.2 “Betting” on a Single Technology (BAD IDEA) ....................... 745
  3.8.3 Section Summary .......................................................... 753

3.9 Ensuring Physical Interoperability ............................................. 753
  3.9.1 Impeding Hierarchical Reuse Is a BAD IDEA ......................... 753
  3.9.2 Domain-Specific Use of Conditional Compilation Is a BAD IDEA ........ 754
  3.9.3 Application-Specific Dependencies in Library Components Is a BAD IDEA ...... 758
  3.9.4 Constraining Side-by-Side Reuse Is a BAD IDEA .................... 760
  3.9.5 Guarding Against Deliberate Misuse Is Not a Goal ................... 761
  3.9.6 Usurping Global Resources from a Library Component Is a BAD IDEA .......... 762
  3.9.7 Hiding Header Files to Achieve Logical Encapsulation Is a BAD IDEA ........ 762
  3.9.8 Depending on Nonportable Software in Reusable Libraries Is a BAD IDEA .... 766
## Contents

3.9.9  Hiding Potentially Reusable Software Is a BAD IDEA .....................................................769
3.9.10 Section Summary ........................................................................................................772
3.10 Avoiding Unnecessary Compile-Time Dependencies ..........................................................773
  3.10.1 Encapsulation Does Not Preclude Compile-Time Coupling .............................................773
  3.10.2 Shared Enumerations and Compile-Time Coupling ..........................................................776
  3.10.3 Compile-Time Coupling in C++ Is Far More Pervasive Than in C .....................................778
  3.10.4 Avoiding Unnecessary Compile-Time Coupling .............................................................778
  3.10.5 Real-World Example of Benefits of Avoiding Compile-Time Coupling ...........................783
  3.10.6 Section Summary .........................................................................................................790
3.11 Architectural Insulation Techniques ...................................................................................790
  3.11.1 Formal Definitions of Encapsulation vs. Insulation .........................................................790
  3.11.2 Illustrating Encapsulation vs. Insulation in Terms of Components ...................................791
  3.11.3 Total vs. Partial Insulation ..............................................................................................793
  3.11.4 Architecturally Significant Total-Insulation Techniques .................................................794
  3.11.5 The Pure Abstract Interface (“Protocol”) Class ..............................................................796
    3.11.5.1 Extracting a Protocol .................................................................................................799
    3.11.5.2 Equivalent “Bridge” Pattern .....................................................................................801
    3.11.5.3 Effectiveness of Protocols as Insulators .....................................................................802
    3.11.5.4 Implementation-Specific Interfaces ..........................................................................802
    3.11.5.5 Static Link-Time Dependencies ..................................................................................802
    3.11.5.6 Runtime Overhead for Total Insulation .....................................................................803
  3.11.6 The Fully Insulating Concrete Wrapper Component .......................................................804
    3.11.6.1 Poor Candidates for Insulating Wrappers ...............................................................807
  3.11.7 The Procedural Interface ...............................................................................................810
    3.11.7.1 What Is a Procedural Interface? .................................................................................810
    3.11.7.2 When Is a Procedural Interface Indicated? .................................................................811
    3.11.7.3 Essential Properties and Architecture of a Procedural Interface .................................812
    3.11.7.4 Physical Separation of PI Functions from Underlying C++ Components ....................813
    3.11.7.5 Mutual Independence of PI Functions .......................................................................814
    3.11.7.6 Absence of Physical Dependencies Within the PI Layer .............................................814
    3.11.7.7 Absence of Supplemental Functionality in the PI Layer ............................................814
    3.11.7.8 1-1 Mapping from PI Components to Lower-Level Components (Using the $z_-$ Prefix) ........................................................................................................815
    3.11.7.9 Example: Simple (Concrete) Value Type .....................................................................816
    3.11.7.10 Regularity/Predictability of PI Names .......................................................................819
    3.11.7.11 PI Functions Callable from C++ as Well as C ............................................................823
    3.11.7.12 Actual Underlying C++ Types Exposed Opaquely for C++ Clients ..........................824
    3.11.7.13 Summary of Essential Properties of the PI Layer ....................................................825
    3.11.7.14 Procedural Interfaces and Return-by-Value ...............................................................826
    3.11.7.15 Procedural Interfaces and Inheritance ........................................................................828
    3.11.7.16 Procedural Interfaces and Templates ..........................................................................829
    3.11.7.17 Mitigating Procedural-Interface Costs .......................................................................830
    3.11.7.18 Procedural Interfaces and Exceptions ........................................................................831
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. 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++. 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 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 \((\text{ellis90})\) 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 \((\text{stroustrup00})\) 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 Nahl, who worked directly with me on readying this book for production. In 2017, I reconnected with my lifelong friend and now wife, Elyse, who tirelessly tracked down copious references and proofread key passages (like this one). By late 2018, it became clear that the amount of work required to produce this book would exceed what anyone had anticipated, and so Pearson retained Lori Hughes to work with me, in what turned out to be a nearly full-time capacity for the better part of 2019. I cannot say enough about the professionalism, fortitude, and raw effort put forth by Lori in striving to make this book a reality in calendar year 2019. I want to thank Lori, Julie, and Greg, and also Peter, Kim, and Debbie, for all their sustained support and encouragement over so many, many years. And this is but the first of three volumes, OMG!

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

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

![Diagram of software organization](image)

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

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

---

1 Open-source code that has been augmented (or forked) to achieve some particular purpose would also fall into this category (e.g., third-party software adapted to use our (polymorphic) memory-allocator model — see Volume II, section 4.10).
entities can then, in turn, be further aggregated into a yet larger body of software, which we call a package group, comprising packages having similar physical dependencies\(^2\) that, taken as a whole, are suitable for independent release, as illustrated in Figure 2-2b.

\(^2\)Note that, while the packages within a group are themselves necessarily internally logically cohesive, such need not be the case for a package group as a whole (see sections 2.8 and 2.9, respectively).
In addition, some of the software that we might need to use could be organized quite differently. For example, we may want to take advantage of certain third-party and open-source libraries, which might not be component-based. We might have our own legacy libraries to use that are also not component-based. These software libraries, of necessity, must come together at a level of aggregation larger than components, as depicted in Figure 2-3.
We generally think of a top-level unit of integration within a large system informally as a “library” whose interface typically consists of a collection of header files in a single directory (e.g., /usr/include) and a single library archive (e.g., libc.a, libc.so) depending on the target platform. We might uniquely refer to this particular architectural entity as a whole as “The C Library” although its internal structure (i.e., how logical content is partitioned among its .o files) is entirely organizational (i.e., not part of its specification or contract; see Volume II, section 5.2) and might vary from one vendor platform to another.

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

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

2.2.1 General Definition of Physical Aggregate

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

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

![Figure 2-4: Logical content aggregated within 15 individual components](image)

2.2.2 Small End of Physical-Aggregation Spectrum

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

\[\text{Figure 2-5: Suite of logically similar yet independent components}\]

---

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

\(^4\) See gamma94.

\(^5\) E.g., `bdlma::MultipoolAllocator`, `bdlma::SequentialAllocator`, and `bdlma::BufferedSequentialAllocator` (see `bde14`, subdirectory `/groups/bdl/bdlma/`).
2.2.3 Large End of Physical-Aggregation Spectrum

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

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

2.2.4 Conceptual Atomicity of Aggregates

**Guideline**

Every physical aggregate should be treated atomically for design purposes.

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

---

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

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

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

**Observation**

The Depends-On relation among aggregates is transitive.

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

2.2.6 Architectural Significance

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

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

---

7 Component Properties 1–3 (sections 1.6.1–1.6.3) and Component Property 4 (section 1.11.1).
A suboptimal implementation might, for example, inadvertently expose a symbol (at the .o level) that was never intended for use outside the UOR. If such unintentional visibility were to occur within a UOR consisting entirely of components, it would likely be due to an accidental violation of Component Property 2 (section 1.6.2) and not a deliberate (and misguided) attempt to provide a secret “backdoor” access point. Repairing such defects would not constitute a change in architecture — especially in this case, since any use of such a symbol would itself be a violation of Component Property 4 (section 1.11.1).

2.2.7 Architectural Significance for General UORs

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

2.2.8 Parts of a UOR That Are Architecturally Significant

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

2.2.9 What Parts of a UOR Are Not Architecturally Significant?

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

---

8 Some methodologies allow for the use of “private” header files (e.g., see Figure 1-30, section 1.4) that are not deployed along with the UOR; our component-based approach (sections 1.6 and 1.11) does not (for good reasons; see section 3.9.7), but does provide for subordinate components (see section 2.7.5).
organization of the library archive that embodies the .o files (corresponding to its .cpp files) comprised by that UOR will have absolutely no effect on client source code. What’s more, changing such insulated details (see section 3.11.1) will not require client code even to recompile.

2.2.10 A Component Is “Naturally” Architecturally Significant

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

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

What ultimately characterizes a component architecturally is governed entirely by its .h file. In Chapter 1, we arrived at the definition of a component as being a .h/.cpp pair satisfying four essential properties. In virtually all cases, this phrasing serves as the definition of a component in C++.9 For completeness, however, we point out that, though this definition is sufficient and practically useful, it is not strictly necessary. The true essential requirement for components in C++ is that there be exactly one .h file and one10 (at least) or more (see below) .cpp files that together satisfy these four essential properties.

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

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

9 More generally, for any given language that supports multiple units of translation (e.g., C, C++, Java, Perl, Ada, Pascal, FORTRAN, COBOL), the physical form of a component is standard and independent of its content.
10 We require that the component header be included in at least one component .cpp file so that we can observe, just by compiling the component, that its .h file is self-sufficient with respect to compilation (section 1.6.1).
11 E.g., to further reduce the size of already tiny programs (such as embedded C) or to break hopelessly large (particularly computer-generated) components into separate translation units of a size manageable for the compiler.
myutil.h and multiple implementation files — e.g., myutil.1.cpp, myutil.2.cpp, and myutil.3.cpp — each uniquely named, but all sharing the component name as a common prefix. Consequently, a program calling only one of the three functions might, under certain deployment strategies (see section 2.15), wind up incorporating only the one .o file corresponding to the needed function. Such nuanced considerations are not relevant to typical development and are most usually relegated to the subdomain of embedded systems.

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

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

2.2.14 Entity Manifest and Allowed Dependencies

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

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

Observation

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

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

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

Figure 2-6: Specifying members and allowed dependencies for aggregates
Unfortunately, the C++ language itself does not support any notion of architecture beyond a single translation unit. Hence, much of the aggregative structure we discuss in this chapter will have to be implemented alongside the language using metadata (see section 2.16). This metadata will be kept locally as an integral part of each aggregate to help guide the tools we use to develop, build, and deploy our software. An abstract subsystem consisting of four second-level aggregates forming three separate (aggregate) dependency levels is illustrated schematically in Figure 2-7.

---

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

---

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

13 A detailed overview of this architectural metadata along with its practical application and how build and other tools might consume it is provided for reference in section 2.16.
2.2.15 Need for Expressing Envelope of Allowed Dependencies

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

2.2.16 Need for Balance in Physical Hierarchy

Observation

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

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

Effective regular decomposition of large systems requires not only hierarchy, but also balance. We choose to model our software development accordingly. Although not strictly necessary, we want each aggregate to comprise entities having similar physical complexity. In particular, we deliberately avoid placing components alongside larger aggregates within a UOR. We find that entities having comparable complexity at each aggregation depth improves comprehension and facilitates reuse.
At each increasing level of physical aggregation, we strive to bring together a significant, but not overwhelming amount of information and engineering at a uniform level of abstraction such that it can be understood and used effectively. As a rule, we would like the relevant schematic detail to correspond to what might reasonably fit on a single 8 1/2 × 11 inch piece of paper as suggested by the complexity of each of the individual diagrams in Figure 2-9. By achieving this balance — much like the chapters and sections within this book — we provide fairly uniformly chunked content, which makes it more convenient to analyze and discuss.

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

(b) Aggregation level II: package of related components

\(^{14}\) Being an American, I have chosen the most common loose-leaf paper size in the United States, as opposed to ones conforming to ISO 216 used by other countries where A4 is the most common (and similar) size (see http://www.papersizes.org/).
2.2.18 Having More Than Three Levels of Physical Aggregation Is Too Many

<table>
<thead>
<tr>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>More than three levels of appropriately balanced physical aggregation are virtually always unnecessary and can be problematic.</td>
</tr>
</tbody>
</table>

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

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

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

\[
\text{source lines} \times \frac{\text{components}}{\text{component}} \times \frac{\text{packages}}{\text{package}} \times \frac{\text{package groups}}{\text{package group}} = \frac{\text{source lines}}{\text{UOR}}
\]

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

2.2.20 UORs Always Have Two or Three Levels of Physical Aggregation

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

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

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

\(^{16}\)The C++ Standard Library residing entirely in the *std* namespace, is itself an example of such a monolithic specification.
2.2.21 Three Balanced Levels of Aggregation Are Sufficient. Trust Me!

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

2.2.22 There Should Be Nothing Architecturally Significant Larger Than a UOR

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

\textsuperscript{17} Having a single, enterprise-wide namespace in which to guard the names within \textit{all} of the components we collectively write is (1) independent of any aspect of specific designs, and (2) a good idea (see section 2.4.6).
Section 2.2  Physical Aggregation

(b) Modeling logical aggregation by individual allowed physical dependencies among UORs

Figure 2-12: Supplanting logical aggregation with allowed physical dependency
2.2.23 Architecturally Significant Names Must Be Unique

<table>
<thead>
<tr>
<th>Design Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>The name of every architecturally significant entity must be unique throughout the enterprise.</td>
</tr>
</tbody>
</table>

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

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

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

— Mike Verschell

\(^{18}\) On April 1, 2019, Mike Verschell became the manager of Bloomberg’s BDE team, replacing its founder (John Lakos) after nearly eighteen rewarding years of applying the methodology described in this book to developing real-world large-scale C++ software. Mike provided the quoted synthesis of his position on unique filenames via personal email.
2.2.24 No Cyclic Physical Dependencies!

**Design Imperative**

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

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

2.2.25 Section Summary

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

\(^{19}\) A collection of interdependent (connected) entities is cyclically dependent if the transitive closure of the binary relation matrix representing direct dependencies between any two entities is not antisymmetric.
2.3 Logical/Physical Coherence

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

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

---

**Guideline**

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

---

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

Avoiding cyclic physical dependencies across aggregate boundaries is not only for the benefit of build tools, it also facilitates human cognition and reasoning. If all that were needed was to have two libraries where the envelope of component dependencies across aggregates was acyclic, then it would suffice to mechanically repartition these components as shown in Figure 2-14. But for software packaging to facilitate human cognition, in addition to being physically acyclic, the logical and physical aspects of a design must remain coherent.
Although the cyclic physical dependencies between the two libraries have been eliminated, the logical and physical designs have diverged. Now, neither logical subsystem is encapsulated by either physical library. As a result, our ability to infer aggregate physical dependencies from abstract logical usage — i.e., at the subsystem level — is lost. That is, if a client abstractly uses either the buyside or sellside logical subsystems, we must either know the details of that usage or otherwise assume an implied physical dependency on both libraries. Just as with cyclic physical dependencies, our ability to reason about logically and physically incoherent designs does not scale; hence, such designs are to be avoided.

Unifying the logical and physical properties of software is what makes the efficient development of large-scale systems possible. Achieving an effective modularization of logical subsystems is not always easy and might require significant adjustment to the logical design of our subsystems (see Chapter 3). As Figure 2-15 suggests, the reworked design might even yield a somewhat different logical model. Achieving designs having both logical/physical coherence and acyclic physical dependencies early in the development cycle requires forethought but is far easier than trying to tweak a design after coding is underway. Once released to clients, however, the already arduous task of re-architecting a subsystem will invariably become qualitatively more intractable, often insurmountably so.
Achieving logical and physical coherence along with acyclic physical dependencies across our entire code base is absolutely essential. In addition to ensuring these important properties, however, we will need a strategy that guarantees not just that the name of each architecturally significant logical and physical entity is unique throughout the enterprise, but that it can also be identified (and its definition located) just from its point of use, without having to resort to tools (e.g., an IDE). The following section addresses how we realize these additional goals in practice.

2.4 Logical and Physical Name Cohesion

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

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

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

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

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

2.4.3 Absurd Extreme of Neither Cohesive nor Mnemonic Naming

Taking this approach to the extreme, we could even have the registry generate unique type names based on a global counter — e.g., T125061, T125062, T125063, and so on. We could do similarly for component names (e.g., c05684, c05685, c05686) and even for units of release (e.g., u1401, u1135, u1564), as illustrated in Figure 2-16. It all works just fine as far as the compiler and linker are concerned. Moreover, physically moving a component from one aggregate to another would have no nominal implications. Human cognition, however, is not served by this approach.
Section 2.4 Logical and Physical Name Cohesion

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

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

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

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

2.4.5 Past/Current Definition of Package

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

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

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

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

---

20 *lakos96*, section 7.1, pp. 474–483
will necessarily be implemented exclusively in terms of components, other kinds of well-reasoned architecturally significant physical entities comprising multiple header files, yet not aggregating components, are certainly possible.21

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

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

### 2.4.6 The Point of Use Should Be Sufficient to Identify Location

<table>
<thead>
<tr>
<th><strong>Guideline</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The use of each logical entity declared at package-namespace scope should alone be sufficient to indicate the component, package, and UOR in which that entity is defined.</td>
</tr>
</tbody>
</table>

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

---

21 Robert Martin is the only other popular author we know of to describe in terms of C++ (previous to lakos96 or otherwise) an even remotely similar concept. In his adaptation of Booch’s Class Categories, which originally were themselves just logical entities (booch94, section 5.1, “Essentials: Class Categories,” pp. 581–584), Martin’s category unites a cluster of classes related by both logical and physical properties. Based on personal (telephone) correspondence (c. 2005), his augmented categories were intended to be significantly larger than a component, but somewhat smaller than a typical package (see Figure 2-11, section 2.2.19), virtually always sporting exactly one class per header (see section 3.1.1); see martin95, “High-Level Closure Using Categories,” pp. 226–231.
of a class, the header file declaring that class, and the UOR implementing that class might all have unrelated names, as illustrated Figure 2-17. Clients reading BondPrice will not be able to predict, from usage alone, which header file defines it, nor which library implements it; hence, global search tools would be required during all subsequent maintenance of client code.

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

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

<table>
<thead>
<tr>
<th>Design Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component files (.h / .cpp) must have the same root name as that of the component itself (i.e., they differ only in suffix).</td>
</tr>
</tbody>
</table>
By their nature, components implemented as .h/.cpp pairs naturally already exhibit some degree of physical name cohesion. Note that as recently as the writing of my first book (1996), however, such was not the case. Due to unreasonable restrictions on the length of names that could be accommodated to distinguish .o files contained in library archive (.a) files of the day, .o files often had to be shortened; hence, an external cross-reference needed to be maintained in order to reestablish the cohesive nature of components.22

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

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

**Design Rule**

**Every component must reside within a package.**

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

---

22 *Iakos96*, Appendix C, pp. 779–813 and, in particular, Appendix C.1, pp. 180–193
rule, however, will fall far short of the potential benefit it seeks to motivate. The intent here is not just to provide a uniform and balanced physical representation of software, but also to craft a hierarchical repository where the contained elements, from a logical as well as a physical perspective, are cohesive and synergistic (see section 2.8.3). Moreover, we want to ensure that each library component we write has a natural and obvious place in the physical hierarchy of our firm-wide repository (see sections 3.1.4 and 3.12).

### Design Rule

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

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

---

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

24 This nomenclature stems from way back before standardization, and we had to use logical package prefixes to implement logical namespaces — e.g., `bget_Point` instead of `bget::Point`. Even with the advent of the namespace construct in the C++98 Standard, we continue to exploit this approach to naming of physical entities and, occasionally, even logical ones — e.g., in procedural interfaces (see section 3.11.7).
Our preference that the names of physical entities (e.g., files, packages, and libraries) not contain any uppercase letters (section 1.7.1) begins with the observation that some popular file systems — Microsoft’s NTFS, in particular — do not distinguish between uppercase and lowercase.\textsuperscript{25} Theoretically, it is sufficient that the \textit{lowercased} rendering of all filenames be unique. Practically, however, having any unnecessary extra degree of freedom in our physical packaging, thereby complicating development/deployment tools, let alone human comprehension, makes the use of mixed-case filenames for C++ source code suboptimal.\textsuperscript{26}

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

\begin{itemize}
\item \textsuperscript{25} With the intent of improving readability (and/or nominal cohesion), it is frequently suggested that we change to allow uppercase letters in component filenames and require them to match exactly the principal class or common prefix of contained classes (see section 2.6), instead of the \textit{lowercased} name as is currently required. We recognize that the readability of multiword filenames can suffer (ironically providing a welcome incentive to keep component base names appropriately concise).
\item \textsuperscript{26} Insisting that our component filenames be rendered in \textit{all_lowercase} also effectively precludes “overloading” on case for logical names, e.g., having both \texttt{DateTimeMap} and \texttt{DateTimeMap} in separate components — which, from a readability standpoint, is something we would probably want to avoid anyway. Imagine trying to communicate such a distinction over a customer-service telephone hotline!
\end{itemize}
between these two distinct dimensions of design, e.g., in component/class diagrams such as the one shown above (Figure 2-18). The utility afforded by this visual distinction within source code and external documents, such as this book, should not be underestimated.

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

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

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

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

28 lakos96, section 7.6.1 (pp. 514–520), and, in particular, Figures 7-21 and 7-22 (p. 519 and p. 520, respectively)
Section 2.4 Logical and Physical Name Cohesion

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

<table>
<thead>
<tr>
<th>Design Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each logical entity declared within a component must be nested within a namespace having the name of the package in which that entity resides.</td>
</tr>
</tbody>
</table>

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

---

29 lakos96, section 7.6.1, pp. 514–520, and in particular Figure 7-21, p. 519
Now that the namespace construct has long since been supported by all relevant C++ compilers, there has been an inculcation toward having concise, unadulterated logical names. Hence, we now (since c. 2005) nest each logical entity within a namespace having the same name as the package containing the component that defines the construct, as shown in Figure 2-20. Our use of logical package namespaces is isomorphic to our original use of logical package prefixes, and therefore consistent with our continued use of physical package prefixes for component filenames to preserve logical and physical name cohesion.
2.4.7 Proprietary Software Requires an Enterprise Namespace

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

**Design Rule**

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

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

---

Note that when namespaces are not appropriate (e.g., functions having extern "C" linkage), we revert back to the use of logical package prefixes (see section 3.11.7).
Figure 2-20, section 2.4.6), we all but eliminate any chance of accidental external collision. And, since all of our components reside within the same enterprise namespace, there is no need or temptation to employ using declarations or directives.\(^{31}\) In the very unlikely event that a collision with external software occurs — even in the presence of using directives — all that is required to disambiguate the collision is to prepend (1) the firm-wide symbol, (2) the third-party product’s symbol, or (3) :: if the third-party code failed to take this precaution.

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

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

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

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

---

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

\(^{32}\) Decentralized registration of packages via package groups (see section 2.9.4) is effective at managing naming conflicts within a single organization. We can, however, easily envisage a world in which source code from multiple enterprises having distinct naming regimes (consistent with our methodology) needs to co-exist within a single code base. Under those circumstances, there might be affirmative value in preventing accidental header-file collisions by proactively adding a very short (e.g., exactly two-character) mutually unique physical prefix (e.g., “bb_”) to each organization’s component names corresponding to (but not necessarily the same as) their respective unique enterprise-wide (logical) namespace names (see sections 2.4.6, 2.4.7, and 2.10.2).
Section 2.4 Logical and Physical Name Cohesion

2.4.8 Logical Constructs Should Be Nominally Anchored to Their Component

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

**Design Rule**

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

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

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

\(^{33}\)Note that this rule may not apply when the external (“client-facing”) component headers are already specified otherwise — e.g., standardized interfaces or established legacy libraries.
Where appropriate, we routinely define outwardly accessible ("public") auxiliary classes, such as iterators, in the same component either by appending to the name of the primary class (e.g., `bdlt::PackedCalendarHolidayIterator`), or else by nesting the auxiliary class within the principal class itself (e.g., `PackedCalendar::HolidayIterator`). Note, however, that some detective work might be unavoidable when operators, inheritance, or user-defined conversion are involved. The rules surrounding the placement of free operators within components are discussed below.

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

**Design Rule**

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

---

34 In practice, the nested iterator type, `PackedCalendar::HolidayIterator`, would likely be a typedef to the non-nested auxiliary iterator class, `bts::PackedCalendarHolidayIterator`, which grants the container private (friend) access (e.g., see section 3.12.5.1). The mandatory colocation of two classes where one grants private access to another is discussed in section 2.6.
Section 2.4 Logical and Physical Name Cohesion

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

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

\(^{35}\) [lakos96], section 2.3.5, p. 77–79, in particular p. 77

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

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

```
typedef int NumElements;
```

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

\(^{38}\) C++11 provides what is known as an `enum class`, which addresses the issue of scoping the enumerators, as well as providing for stronger type safety. Note that all enumerations in C++11 allow their underlying integral type to be specified and, unlike C++03, thereby form what is known as a `complete type`, enabling them to be declared and used locally (i.e., without also specifying the enumerators). The ability to elide enumerators can constitute what is sometimes referred to in tort law as an “attractive nuisance” in that, unless the elided enumeration is supplied by a library in a header separate from the one containing its complete definition, a client wishing to insulate itself from the enumerators would be forced to declare the enumeration locally in violation of Component Property 3 (section 1.6.3).
The justification for avoiding free functions, except operator and operator-like “aspect” functions, which might benefit from argument-dependent lookup (ADL), derives from our desire to encapsulate an appropriate amount of logically and physically coherent functionality within a nominally cohesive component. While classes are substantial architectural entities that are easily identifiable from their names, individual functions are generally too small and specific for each to be made nominally cohesive with the single component that defines them, as in Figure 2-22a.\textsuperscript{39}

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

\begin{verbatim}
// xyza_roundtowardzero.h
namespace xyza {
    double roundTowardZero(double value);
}  // close package namespace

// xyza_mathutil.h
namespace xyza {
    double roundTowardZero(double value);
    double factorial(double value);
}  // close package namespace
\end{verbatim}

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

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

\textsuperscript{39}Given that we virtually always open and close a package namespace exactly once within a component (see section 2.5), we choose not to indent its contents, thereby increasing usable real estate given a practical maximum line length (e.g., 79) suitable for efficient reading, printing, side-by-side comparison, etc. (see Volume II, section 6.15).
Section 2.4  Logical and Physical Name Cohesion

We therefore generally avoid declaring free (nonoperator) functions at package-namespace scope, and instead achieve both nominal logical and physical cohesion by grouping related functionality within an extra level of namespace matching the component name using static methods within a struct (Figure 2-22e), which we will consistently refer to as a utility...
There are many advantages of using a `struct` (e.g., Figure 2-22e) over a third level of `namespace` (e.g., Figure 2-22d) for implementing a utility are summarized in Figure 2-23.

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

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

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

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

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

6. Unlike a `namespace`, a C-style function in a `struct` does not participate in Argument-Dependent Lookup (ADL), thereby avoiding potentially large overload sets, which could needlessly affect compile-time performance and possibly introduce unanticipated (perhaps even latent) ambiguity, or — much worse — invoke the wrong function.

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

---

**Figure 2-23:** Prefer `struct` to `namespace` for aggregating “free” functions.
Design Rule

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

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

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

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

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

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

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

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

```cpp
template <class TYPE>
std::ostream& operator<<(std::ostream& lhs,
                        const std::vector<TYPE>& rhs);
```
(along with an appropriate definition) in its header for general use by clients. It is not hard to imagine that component `your_stuff` might do so as well. Now consider what happens when `their_stuff.cpp` includes both `my_stuff.h` and `your_stuff.h`. The inevitable result is multiply defined symbols!44

![Diagram](image)

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

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

---

44 Because the offending operator is a template, which has dual bindage (section 1.3.4), it is entirely possible that the duplicate definitions will go unnoticed by either the compiler or the linker for quite some time — that is, until the compiler can see the two template definitions side-by-side in a single translation unit. Had the construct instead had external bindage, such as an ordinary function or an explicit instantiation, merely linking the two components into the same program would have been sufficient to expose the incompatibility.
As illustrated in Figure 2-26, providing an output operator on a type `my::Type` — or conceivably even on a `std::vector<my::Type>` — in component `my_type` is perfectly fine. The general design concept being illustrated here is to follow the teachings of the philosopher Immanuel Kant and avoid doing those things that, if also done by others, would adversely affect society (see section 3.9.1). By adhering to this simple rule for operators, we ensure that (1) we know where to look for each operator, and (2) operator definitions will not be duplicated (and therefore cannot conflict at higher levels in the physical hierarchy).
Figure 2-26: Overloading free operators on types within the same component

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

1. [Suboptimal] Arbitrarily choose one of the components to be at a higher-level and place the free operator there, as in Figure 2-27b (thus introducing additional physical dependency for one of the components).

2. [Preferred] Create a utility class in a separate component, as in Figure 2-27c, and define one or more nonoperator functions nested within a struct that serves the same purpose (see section 3.2.7). Note that it is never appropriate to escalate (see section 3.5.2) co-dependent free operators to a separate component.

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

45 We note that the C++ streaming operators and Boost.Spirit are (rare) arguably plausible counter-examples; still, we maintain that heterogeneous equality comparison operators across disparate user-defined value types (see Volume II, section 4.1), such as Square and Rectangle (Figure 2-27), remain invariably misguided for entirely different reasons (see Volume II, section 4.3).
Section 2.4  Logical and Physical Name Cohesion

(a) Addressing placement of heterogeneous operators

```
bool operator==(const xyza::Square& lhs, const xyza::Rectangle& rhs);
bool operator==(const xyza::Rectangle& lhs, const xyza::Square& rhs);
```

(b) By introducing additional dependencies [SUBOPTIMAL]

```
namespace xyza {
    struct SquareRectangleUtil {
        static bool areEqual(const Square& square, const Rectangle& rectangle);
        static bool areEqual(const Rectangle& rectangle, const Square& square);
    };
    // ...
}  // close package namespace
```

(c) By escalating and replacing with static methods of a struct [PREFERRED]

```
namespace xyza {
    struct SquareRectangleUtil {
        static bool areEqual(const Square& square, const Rectangle& rectangle);
        static bool areEqual(const Rectangle& rectangle, const Square& square);
    };
    // ...
}  // close package namespace
```

Figure 2-27: Implementing “free operators” referring to multiple peer types
2.4.10 Package Prefixes Are Not Just Style

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

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

The problem, identified by Figure 2-29, can easily arise in practice. Consider the design of a single package that is intended to contain everything that is directly usable by clients of a multipackage subsystem. If this presentation package (\texttt{subc}) defines both protocol (i.e., pure abstract interface) classes (which are inherently very low level) and wrapper components (which are inherently very high level), it will not be possible to interleave components from a separate implementation package (\texttt{subim}).\footnote{For complex subsystems, the implementation components represented here as a single package \texttt{subim} may appropriately span many packages at several different levels; however, the basic idea remains the same.}
Figure 2-29: Acyclic component hierarchy; cyclic package hierarchy (BAD IDEA)

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

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

---

\(^{47}\)A *wrapper* is a *facade* that allows clients to manipulate objects (typically of some other type) without providing direct programmatic access to those objects (see sections 3.1.10 and 3.11.6).
(e.g., `subv_comp1`) that define protocol and other *vocabulary* types (see Volume II, section 4.4) exposed programmatically through the wrapper interface.\textsuperscript{48}

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

\textsuperscript{48} See the *escalating encapsulation* levelization technique (section 3.5.10).
Section 2.4  Logical and Physical Name Cohesion

(a) Parallel protocol and concrete vocabulary-type packages (BAD IDEA)
Figure 2-31: Alternative packaging strategies

2.4.11 Package Prefixes Are How We Name Package Groups

Although packages, being architecturally significant aggregates, have unique names (and namespaces), it is often advantageous to bundle packages having similar purposes and/or similar envelopes of physical dependency into a larger, logically and physically coherent, nominally cohesive aggregate. We could make a big deal about this issue (and perhaps we should, given its importance). Instead we will avoid the drama and just make our point: The first three letters of a package name identify the physically cohesive package group in which a grouped package resides.
The reason for this simple approach is, well, simple (see section 2.10.1): We simply must have an ultra-efficient way to specify the package group and package of each component and class in order to obviate noisome and debilitating using directives and declarations (see section 2.4.12). The choice of three letters (as opposed to, say, two or four) is simply an engineering trade-off. This simple, concise, and effective approach to naming package groups is illustrated in Figure 2-32. We will revisit our package-naming rules (in much greater depth) in section 2.10.

Figure 2-32: Logically and physically cohesive package group
2.4.12 using Directives and Declarations Are Generally a BAD IDEA

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

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

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

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

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

inline
static void insertAfterLink(Link *node, Link *newNode)
{
    BSLS_ASSERT(node);
    BSLS_ASSERT(newNode);
    newNode->next = node->next;
    newNode->prev = node;
    node->next = newNode;
    if (newNode->next) {
        newNode->next->prev = newNode;
    }
}

// ... (See Volume II, section 6.8.)

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

\(^{50}\)\textit{sutter05}, item 59, pp. 108–110
static void communicate(Relay *relay)
{
    static Callback myCallback;
    if (relay->isOperational()) {
        relay->setForwardCallback(&myCallback);
    } else {
        Log::singleton().write("Life is like a box of chocolates...");
    }
    // ...
}

Figure 2-34: using directives/declarations can be included! (BAD IDEA)
Neither using directives nor using declarations are permitted to appear outside function scope within a component.

No matter what, we must forbid any using directives or declarations in header files outside of function scope. Perhaps some advocates of using in headers might not yet have realized that the incorporation of names from one namespace, A, into another, B, does not end with the closing brace of B into which names from A were imported, but remain in B until the end of the translation unit. Consequently, using directives or declarations are sometimes used (we should say horribly misused) in header files when declaring class member data and function prototypes to shorten the names of types declared in distant namespaces. And, in library code, using is generally best avoided altogether. If used there at all, a using declaration (not directive) — whether employed to enable ADL (e.g., for a free aspect function, such as swap), or merely as a compact alias (e.g., as an entry into a dispatch table) — should appear only within a very limited lexical context, i.e., function (or block) scope.

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

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

Alisdair Meredith notes (via personal email, 2018) that, when a base class is a template, the set of overloads to forward is an open set. Accidental breakage can occur when a design requires that each of the overloads be exposed manually. When the intent is to perfectly forward an overload set from a base class, a using declaration is a clear statement of that design intent. Nonetheless, our recommended approach is to avoid such uses of (typically structural) inheritance (see Volume II, section 4.6), preferring the more compositional Has-A (section 1.7.4) approach to layering (see section 3.7.2) instead.

That said, exceptional cases do exist. Alisdair Meredith further points out (again, via personal email, 2018) that we ourselves have, on occasion, been known to introduce a base class having fewer template parameters, and then use structural inheritance and using declarations to expose that functionality as the public interface. If we were now to replace using declarations with, say, inline forwarding functions, we would negate the intended effect of reducing template-induced code bloat (see Volume II, section 4.5).
Instead, we must use the package-qualified name of each logical entity not local to the enclosing package. For this reason, we will want to ensure that widely used ("package") namespace names, like std, are very short indeed.

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

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

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

---

55 Local typedefs have historically been effective at addressing long names in data definitions and function prototypes due to specific template instantiations:

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

We recognize that C++11 offers using as a syntactic alternative, and that thoughtful (discriminating) use of auto can also help eliminate redundant (or otherwise superfluous) explicit type information in source code. See lakos21.
of the same logical entity to co-exist for a period of time as clients rework their code to refer to the new component before the original one is finally removed.56

2.4.13 Section Summary

In summary, our rigorous approach to cohesive naming — packages, components, classes, and free (operator) functions — not only avoids collisions, it also provides valuable visual cues within the source code that serve to identify the physical location of all architecturally significant entities. Experience shows that human cognition is facilitated by such visual associations. In turn, this nominal cohesion reinforces the even more critical requirement of logical/physical coherence (section 2.3). Hence, logical and physical name cohesion across related architecturally significant entities is an integral part of our component-based packaging methodology.
Symbols
<> (angle brackets), 202–203, 344, 369–370, 433, 490
.. (ellipses), 238
== (equality) operator, 221–222, 882
!= (inequality) operator, 221–222, 511
() (parentheses), 652
+ (plus sign), 431–432
" (quotation marks), 202–203, 344, 369–370, 433, 460, 490
_ (underscore)
  in component names, 53, 304, 381–383, 487, 938–939
  conventional use of, 371–377
  extra underscore convention, 372–377, 561, 591, 771, 939
  in package names, 425
A
AA (allocator-aware) objects, 807–808
absEqual method, 34
abstract data types (ADTs), 192
abstract factory design pattern, 556–557
abstract interfaces, 498–499, 526
abstract syntax tree (AST), 557
Account class, 717–722
Account report generator, 37–40
ACE platform, 719
active library development, 811
acyclic dependencies. See also cyclic dependencies
  component collections, 93–95
  components, 362–370
defined, 936
  levelization and, 251–256, 602
libraries, 149–151, 417–421
package groups, 411–413
package prefixes, 322–326, 937
acyclic logical/physical coherence, 296–297
Ada, 125
adapters, 601, 736, 754–758, 803
adaptive allocation, 783
addDaysIfValid function, 844
additive values, 839, 881
addNode function, 667, 673
addresses, program-wide unique, 163–166
ADL (argument-dependent lookup), 200, 314
ADTs (abstract data types), 192
advanceMonth function, 878–879
aggregation. See physical aggregation
agile software development, 29–30, 433
aliases, namespace, 200
all-lowercase notation
  component names, 304–305, 938
  package group names, 423–424, 939
  package names, 424–426, 939
  procedural interface names, 819–820
allocate method, 699, 778
Allocator protocol, 860, 902
allocators
  Allocator protocol, 860, 902
  allocator-aware (AA) objects, 807–808
default, 860
factories, 505
memory allocation, 808
open-source implementation, 785
stateful, 808
allowed dependencies
  defined, 936
  entity manifests and, 281–284, 936
package groups, 408–413, 939–941
physical aggregates, 300, 938, 942
all-uppercase notation, 371–372, 938
alphabetization of functions, 845
amortized constant time, 534
angle brackets (< >), 202–203, 344, 369–370, 433, 490
ANSI-standard Gregorian calendar, 886
anticipated client usage, modularization and, 523–528
a.out filename, 131
applications. See also compilation; library software; linkage
agile software development, 433
application-specific dependencies, 758–760, 941
creating, 126–128
defined, 6
development framework for, 433–437, 491
“Hello World!”, 125–126
“ill-formed”, 692–693
library software compared to, 5–13
naming conventions, 435–436, 940
programs in, 434
reusability of, 6–13
structure of, 125–126
top-down design, 6–7
ar archiver program, 145
architecture. See also insulation; metadata
architectural entities, 274
coarsely layered, 22–23
finely graduated, granular, 23–27
interpreters, 384–385
lateral
CCD (cumulative component dependency), 723, 727–732
versus classical layered architecture, 723–726
construction analogy, 723
correspondingly layered architecture, 729
defined, 223
versus inheritance-based lateral architectures, 732–738
layered clients, 498–499
light versus heavy layering, 728–729
mail subsystem, 599
overview of, 722–723
private inheritance versus, 225, 332
protocols and, 802
purely compositional designs, improving, 726–727
summary of, 738–739, 909, 917–918
testing, 738
SOAs (service-oriented architectures)
cyclic physical dependencies and, 519
insulation and, 833
procedural interfaces compared to, 715
archives. See library software
area, polygons, 537–539
argument-dependent lookup (ADL), 200, 314
asDatetimeTz method, 849
as-needed linking, 145
aspect functions, 311, 335, 423, 483, 839, 937–938
Aspects subcategory, 841
assembly code, 129
Assert class, 904
AST (abstract syntax tree), 557
atomicity. See also components
atomic units, 48
libraries, 277
object files (.o), 131–134
physical aggregates, 277
automatic storage, 162
autonomous core development team, 98–100
auxiliary date-math types, 878–881
axioms, 437

B
balance, in physical hierarchy, 284–287, 290
ball (BDE Application Library Logger), 599, 761
banners, 335–336
Bar class, 156–157, 355–359
BAS (Bloomberg Application Services), 833
base classes, 331
base names, 292, 310, 372, 936
Base64Encoder class, 521
BaseEntry class, 141
Basic Business Library Day Count package, 570–574
Basic Service Set. See bss segment (executables)
BDE Application Library Logger (ball), 599, 761
BDE Development Environment, 839, 840
BDE Standard Library (bsl), 404
BDEX streaming, 839–848, 898, 902
bdex_StreamIn protocol, 839
bdex_StreamOut protocol, 839
bdlmam_pool component, 788
bdlt_testcalendarloader component, 455
Bear Stearns, 15, 89, 783
benign ODR violations, 160, 195, 264
“hetting” on single technology, 745–753
“Big Ball of Mud” design, 5
bimodal development, 95
binary relations, transitive closure on, 259
bindage
   declaring in header (.h) files, 214–216, 344–345
   external/dual, 163, 935
   internal, 805, 935
   overview of, 160–162, 263
BitArray type, 895–898
bitset, 896
BitStringUtil struct, 898
BitUtil struct, 897–898
black-box testing, 445
Blackjack model, 655–660
blockSize parameter (Pool class), 785
Bloomberg Application Services (BAS), 833
boilerplate component code, 334
“boiling frog” metaphor, 776
Booch’s Class Categories, 301
Boost’s C++98 concepts library, 234
Boost.Test, 456
Box class, 604–609
Breitstein, Steven, 906
bridge pattern, 801
brittleness, 15–17, 116, 781
Brooks, Fred, 4, 88
brute-force solutions, 64–70, 668
bsl (BDE Standard Library) package group, 404
bslma::Allocator, 902
bsls_assert component, 904
bss segment (executables), 131–132
budgeting, 3–5, 115
build process
   build requirements metadata, 475–476, 493
      example of, 131–134
   link phase, 131–132, 260
   object files (.o), 131–134
   overview of, 129–134
   preprocessing phase, 129–130
   software organization during, 462
   translation phase, 129–130, 132
build requirements metadata, 475–476, 493
build-time behavior, link order and, 151
business-day functionality, date/calendar subsystem
   adding to Date class, 715–717
   holidays, 855, 859
   locale differences, 854
   requirements for, 837
Business-Object-Loaders subsystem, 733
ByteStream class
   brute-force solutions based on redundancy, 668
   standardizing on abstract ByteStream interface class, 668–669
standardizing on ByteStream concept, 669–671
standardizing on single concrete ByteStream class, 665–667

C
C language, 125, 811–812
The C++ Programming Language (Stroustrup), 870–871
cache
calendar-cache component, 454–456
date/calendar subsystem
CacheCalendarFactory interface, 867–871
CalendarCache class, 861–867
software reuse and, 85–86
CacheCalendarFactory interface, 867–871
calculateOptimalPartition, 60, 67
calendar and date subsystem. See date/calendar subsystem
Calendar class, 895–899
Calendar type, 855
CalendarCache class, 861–867
CalendarFactory interface, 867–871
CalendarLoader interface, 862–867
CalendarService class, 715
CalendarUtil structure, 883
callables, 639
callbacks
concept
brute-force solutions based on redundancy, 668
defined, 664–665
standardization on single concrete ByteStream class, 665–667
standardizing on abstract ByteStream interface class, 668–671
support for, 664
data, 640–643
function
cyclic rendering of Event/EventMgr subsystem, 647–648
defined, 643–644
disadvantages of, 651
eliminating framework dependencies with, 649–651
function callbacks in main, 644–647
functor
defined, 651
eliminating framework dependencies with, 652–654
stateless functors, 654–655
overview of, 639
protocol
Blackjack model, 655–660
logger-transport-email example, 655–660
summary of, 915
calling procedural interface functions, 823–824
.cap files, 433
capabilities metadata, 476
capital, software
autonomous core development team, 98–100
benefits of, 91–98
defined, 89
demotion process, 95
hierarchically reusable software repository, 108–109
in-house expertise, 107–108
intrinsic properties of, 91–92
mature infrastructure for, 106–107
motivation for developing, 89–90
origin of term, 89
overview of, 86–98
peer review, 90–91
quality of, 110–114
recursively adaptive development, 100–105
return on investment, 86–88
summary of, 120–121
Cargill, Tom, 643
categories, 564
CC compiler, 136
CCD (cumulative component dependency) defined, 727–730
example of, 730–732
minimizing, 727–729
CCF (contract-checking facility), 664
Cevelop, 258
Index

Channel class, 230, 745–753
ChannelFactory class, 745–753
channels
  channel allocator factories, 505
  channel allocators, 505
  Channel class, 230, 745–753
  channel protocols, 505
  ChannelFactory class, 745–753
defined, 505
CharBuf class, 667
charter, package, 502
chunkSize parameter (Pool class), 785, 788
Circle class, 798
cl compiler, 136
Clang, 259, 328
classes. See also enumerations; protocols
  Account, 717–722
  adapter, 736
  Allocator, 785
as alternative to qualified naming, 198–201
  Assert, 904
  Bar, 156–157, 355–359
  base classes, 331
  Base64Encoder, 521
  BaseEntry, 141
  Booch’s Class Categories, 301
  Box, 604–609
  ByteStream
    brute-force solutions based on redundancy, 668
    standardizing on abstract ByteStream
      interface class, 668–669
    standardizing on ByteStream concept, 669–671
    standardizing on single concrete
      ByteStream class, 665–667
  Calendar, 895–899
  CalendarCache, 861–867
  CalendarService, 715
categories of, 564
  Channel, 230, 745–753
  ChannelFactory, 745–753
  CharBuf, 667
Circle, 798
colocation
  component-private classes, 561–564
criteria for, 501, 522–527, 555–560,
    591, 941
day-count example, 566–576
  mutual collaboration, 555–560, 941
  nonprimitive functionality, 541, 941
  single-threaded reference-counted functors
    example, 576–591
  subordinate components, 564–566
  summary of, 591–592, 912–914, 941
template specializations, 564
CommonEventInfo, 616–617
component-private
defined, 371, 937
example of, 378–383
  identifier-character underscore (_), 371–377
  implementation of, 371
  modules and, 371
  summary of, 384, 486–487
  concrete, 498–499
Container_Iterator, 380
Date
  business-day functionality, 715–717,
    854–855
class design, 838–849
day-count functions in, 567
  hidden header files for logical
    encapsulation, 763–764
  hierarchical reuse of, 886–887
  indeterminate value in, 842
  nonprimitive functionality in, 709–714
  physical dependencies, 740–744
  value representation in, 887–895
DateSequence
component/class diagram, 508–509
  open-closed principle, 511
  single-component wrapper, 509–510
DateSequenceIterator, 509–510, 515
DateUtil, 610–611, 742–743
Default, 785
Dstack, 774–775
Edge, 673–674
dumb-data implementation, 629–633
factoring, 675–676
manager classes, 673–674
opaque pointers and, 625–629
dumb-data implementation, 629–633
factoring, 675–676
manager classes, 673–674
opaque pointers and, 625–629
enum, 313
Event, 624
EventQueue, 615–618
Foo, 156, 355
FooUtil, 179–183
grouping functionality of, 841
inheritance
constrained templates and, 230–233
equivalent bridge pattern, 801
inheritance-based lateral architectures, 732–738
private, 692
procedural interfaces, 828–829
public, 359–362
relationships and, 234
Link, 671
List, 671–673
local declarations, 507, 594, 794
MailObserver, 663
manager, 671–674
MonthOfYear, 878
MySystem, 231
nested
constructors, 375
declaring, 375–377
defining, 373, 940
protected, 377
Node, 625
dumb-data implementation, 629–633
factoring, 675–676
manager classes, 673–674
opaque pointers and, 625–629
Opaque, 168
OraclePersistor, 736
OsUtil, 742–743
package namespace scope, 312–321, 483, 938, 940
PackedCalendar, 859–861, 900–901
Persistor, 733–738
Point, 169–170, 816–824
PointList, 239–241
Polygon, 35
“are-rotationally-similar” functionality, 541–544
flexibility of implementation, 535–537
implementation alternatives, 534–535
interface, 545–552
invariants imposed, 531
iterator support for generic algorithms, 539–540
nonprimitive functionality, 536–537, 541
performance requirements, 532–533
Perimeter and Area calculations, 537–539
primitive functionality, 533–534, 540
topologicalNumber function, 545
use cases, 531–532
values, 530
vocabulary types, 530–531
Pool, 778–783
inline methods, 781–783
partial insulation, 782
replenishment strategy, 784–789
PricingModel, 758–759
ProprietaryPersistor, 733
PubGraph, 685
Rectangle, 604–609, 798
Registry, 145
RotationalIterator, 544
salient attributes, 515
shadow, 516–517
Shape, 795–798
ShapePartialImp, 799–800
ShapeType, 808
Stack, 49
StackConstIterator, 49
templates, 179–183
TestPlayer, 659
TimeSeries, 509–510
component/class diagram, 508–509
Index

hidden header files for logical encapsulation, 763–765
wrappers, 512–516
TimeSeriesIterator, 508–510
unconstrained attribute, 610
classical layered architecture, 723–726
classically reusable software, 18–20, 116
client-facing interfaces, name cohesion in, 313
clients, layered, 498–499
closure, 528
course dependencies, predefining with package groups, 417–419
coarsely layered architecture, 22–23
Cobol, 125
code bloat, 561, 780
coelected upgrades, 32
coherence, logical/physical
  overview of, 294–297
  package groups and, 414–417
  summary of, 482–484
cohesion, name, See logical/physical name cohesion
coincidental cohesion, 395–396
collaborative logical relationships
  In-Structure-Only, 227–230
  Uses-In-Name-Only, 226–227
collaborative software, reusability in, 14–20, 116
colocation
  component-private classes, 561–564
criteria for
  cyclic dependency, 557, 591
  “flea on an elephant,” 559–560, 591
  friendship, 556–557, 591
  overview of, 522–527, 555–560, 591, 941
  single solution, 557–559, 591
  substantive nature of, 501
day-count example, 566–576
  bblde package implementation, 570–574
  ISMA 30/360 day-count convention, 567
  library date class, 567
  package implementation, 575–576
  protocol class implementation, 573–575
  PSA 30/360 day-count convention, 567
  single-component implementation, 568–570
mutual collaboration, 555–560, 941
nonprimitive functionality, 541
single-threaded reference-counted functors
  example
    aggregation of components into packages, 586–589
    event-driven programming, 576–586
    overview of, 555–576
    package-level functor architecture, 586–589
    subordinate components, 564–566
    summary of, 591–592, 912–914, 941
    template specializations, 564
commands. See also functions and methods
dumpbin, 133
nm, 133
CommonEventInfo class, 616–617
compare function, 172–174
competition, perfect, 87
compilation, 259–260. See also library software
  build process, 129–134
  compiler programs, 136
  compile-time, avoidance of, 773
  compile-time dependencies, 239, 359–362
    avoiding unnecessary, 778–783
    defined, 936
    encapsulation, 773–776
    pervasiveness of, 778
    real-world example, 783–789
    shared enumerations, 776–777
    summary of, 790, 920
  compile-time polymorphic byte streaming, 415
cost of, 773
declarations
  aspect functions, 335
  consistency in, 194–201
  defined, 153–154
  definitions compared to, 154–159
  forward, 358–359
  inline functions, 778–783, 939
local, 507, 594, 794
at package namespace scope, 312–321
program-wide unique addresses, 163–166
pure, 188, 358
summary of, 188–190, 261–265
typedef, 168, 313
using, 328–333
visibility of, 166–170
defined, 129
definitions
compiler access to definition’s source code, 166–168
declarations compared to, 154–159
declaring in header (.h) files, 212–214, 344
defined, 153–154
entities requiring program-wide unique addresses, 163–166
global, 475, 762
local, 475
ODR (one-definition rule), 158, 185–186, 262–264
self-declaring, 155, 188, 261
summary of, 188–190, 261–265
visibility of, 166–170
domain-specific conditional, 754–758
header (.h) files
architectural significance of, 280–281
build process, 129–134
in course-grain modular programs, 192
declaration consistency in, 194–201
external bindage, 214–216, 344–345
external linkage, 212–214, 344–345
in fine-grained modular programs, 193–194
as first substantive line of code, 210–212, 343–344
hiding for logical encapsulation, 762–765, 942
macros in, 212
modularization of logical constructs, 214
overview of, 48, 119, 190–201
pqrs_bar.h, 355–359
private, 192, 279, 352
purpose of, 128–129, 190–191
source-code organization, 333–336, 938–939
structs in, 9
stylistic rendering within, 463–464
summary of, 264–265, 937–939
unique names, 460
in unstructured programs, 191–192
#include directives
component design rules, 359–362, 940
component functionality accessed via, 257–259, 346
external include guards, 205–208, 353
hierarchical testability, 447, 449, 940
internal include guards, 203–209, 353, 939
removing unnecessary, 258
source-code organization, 334
summary of, 265
text and use, 201–203, 942
transitive includes, 227, 359–360, 486, 605–609
linkage
class templates, 179–183
compiler access to definition’s source code, 166–168
cost entities, 188
enumerations, 170–171
explicit specialization, 174–179
extern template functions, 183–185
external, 158, 262–263
function templates, 172–179
how linkers work, 162–163, 260
inline functions, 166–168, 171–172, 177
internal, 159, 262–263
linkers, 131–132, 260
logical nature of, 159
namespaces, 186–188
ODR (one-definition rule), 185–186
overview of, 153
program-wide unique addresses and, 163–166
summary of, 188–190, 261–265
type safety, 127–128
object files (.o)
atomicity of, 131–134
build process, 131–134
naming conventions, 131
sections, 135, 138–139
static initialization, 152
undefined symbols in, 133, 146
unique names, 460
weak symbols in, 138–139
zero initialization, 131–132
recompilation, 773
“singleton” registry example, 141–146
complete functionality, 528
completeness, 528, 545, 554, 910, 941
component-private classes, 561–564
defined, 371, 937
example of, 378–383
implementation of, 371
modules and, 371
summary of, 384, 486–487
components. See also date/calendar subsystem;
dependencies; header (.h) files;
implementation (.cpp) files; physical
design
advantages of, 20
architectural significance of, 280–281, 936
as atomic unit of physical design, 48
bdlma_pool, 788
bsls_assert, 904
completeness, 528, 545, 554, 910, 941
cyclically dependent, 592–594
defined, 2, 47–48, 117, 209–210, 244, 936
design rules
component properties and, 342–346
cyclic physical dependencies, 362–370, 939
#include directives, 359–362, 939–940
inline functions, 354, 939
internal include guards, 353, 939
logical constructs, anchoring to components, 346–353
regularity in, 353
runtime initialize of file- or namespace-
scope static variables, 354–359, 939
summary of, 485–486, 938–940
drivers associated with, 441–445
as fine-grained modules, 498
focused purpose, need for, 527
hierarchical testability requirement, 437
allowed test-driver dependencies across packages, 451–454, 940
associations among components and test drivers, 441–445
black-box testing, 445
dependencies of test drivers, 445–447, 940
directory location of test drivers, 445, 940
fine-grained unit testing, 438
import of local component dependencies, 447–451
#include directives, 447, 449, 940
minimization of test-driver dependencies on external environment, 454–456
need for, 439–441, 940
summary of, 458–459, 491–492
uniform test-driver invocation interface, 456–458, 941
“user experience,” 458, 941
white-box knowledge, 445
implementation, 677
inherently primitive functionality, 528–553
insulating wrapper, 687
leaf, 251–253, 573–574
logical constructs, anchoring to, 311–312, 346–353
logical versus physical view of, 49–55
minimalism, 528, 554, 910
mocking, 526, 659, 733
my_stack example, 49–53
naming conventions, 53, 301–309, 937–939, 942
package-local (private), 769–772, 942
physical uniformity, 46–57
  developer mobility and, 47
  importance of, 46–47
placement of, 395–396
primitiveness
  closure and, 528
  defined, 911
  manifestly primitive functionality, 528–529, 942
  in Polygon example, 533–534
  quick reference, 941
properties of
  external bindage, 214–216, 344–345
  external linkage, 212–214, 344
  header as first substantive line of code, 210–212, 343–344
  modularization of logical constructs, 214
  overview of, 210–216, 280, 342–346
  summary of, 265–266, 485
relationships
  Depends-On, 218, 237–243, 278
  “inheriting,” 234
  In-Structure-Only, 227–230
  Is-A, 219, 243–251
  Uses-In-Name-Only, 226–227
  Uses-In-The-Implementation, 221–225, 243–251
  Uses-In-The-Interface, 219–220, 243–251
  scope of, 55–56
size of, 508
source-code organization, 333–342, 938
standard, 111
subordinate, 372, 486–487, 564–566, 591, 937, 939
sufficiency, 528, 554, 910
suffixes, 553
summary of, 118–119
testability of, 49
testcalendarloader, 455
text-partitioning optimization problem
  brute-force recursive solution, 64–70
  component-based decomposition, 60–64
  dynamic programming solution, 70–76
exception-agnostic code, 62
exception-safe code, 62
lookup speed, 79–83
probability of reuse, 84–86
real-world constraints, 86
reuse in place, 76–79
vocabulary types, 85
as units of deployment, 47, 555
composition. See layered architectures
concepts
  concept callbacks
    brute-force solutions based on redundancy, 668
    defined, 664–665
    standardizing on abstract ByteStream interface class, 668–669
    standardizing on ByteStream concept, 669–671
    standardizing on single concrete ByteStream class, 665–667
    support for, 664
day-count example, 573–575
defined, 229
history of, 236
concrete classes, 498–499
conditional compilation, domain-specific, 754–758, 941
conditional runtime statements, 756
conforming types, 172
const references, 619, 622
  const correctness, 624
  linkage, 188
  named constants, 843
  non-const access, 624
constrained templates, interface inheritance and, 230–233
constructors, nested classes, 375
consume method, 699
Container_Iterator class, 380
context, 577
continuous refactoring, 14, 419, 461, 634
contract-checking facility (CCF), 664
contracts, 9, 274
Coordinated Universal Time (UTC), 849
correctness, const, 624
correspondingly layered architecture, 729
costs
  compilation, 773
  low-level cycles, 599
  procedural interfaces, 830–831
  schedule/product/budget trade-offs, 3–5
coupling, compile-time. See also dependencies
  avoiding unnecessary, 778–789
  encapsulation, 773–776
  pervasiveness of, 778
  real-world example, 783–789
  reducing, 741
  shared enumerations, 776–777
  summary of, 790, 920
covariant return types, 359
  __cplusplus preprocessor symbol, 823–824
  .cpp files. See implementation (.cpp) files
cracked plate metaphor, 14–20, 116
cumulative component dependency (CCD)
  defined, 727–730
  example of, 730–732
  minimizing, 727–729
CurrentTimeUtil struct, 849–853
cyclic dependencies. See also levelization
  techniques
  avoidance of, 592–601
  colocation, 557, 591
  components, 592–594
  cyclically realization of entity/relation
    model, 594–596
  dependency evolution over time, 597–601
  Google’s approach to, 519
  physical design thought process, 505–507
  subsystems, 596–597
  summary of, 601, 914–915
components, 362–370
hierarchical testability requirement
  allowed test-driver dependencies across
    packages, 451–454, 940
  associations among components and test
    drivers, 441–445
black-box testing, 445
dependencies of test drivers, 445–447, 940
directory location of test drivers, 445, 940
fine-grained unit testing, 438
import of local component dependencies, 447–451
#include directives, 447, 449, 940
minimization of test-driver dependencies
  on external environment, 454–456
  need for, 439–441, 940
  overview of, 437
  summary of, 458–459, 491–492
  uniform test-driver invocation interface, 456–458, 941
“user experience”, 458, 941
white-box knowledge, 445
library software, 146–151
logical/physical coherence, 294–295
packages
  overview of, 394–395, 939–941
  package groups, 411–413
  package prefixes, 322–326
physical design and, 45
undesirability of, 292–293
cyclic rendering of Event/EventMgr subsystem, 647–648
cyclically dependent design, 592

D
d_freeList_p function, 776, 781
d_mechanism_p pointer, 699
DAG (directed acyclic graph), 251–252
data, dumb, 629–633, 915
data callbacks, 640–643
data members, number of, 837
Date class, 887–895
  business-day functionality, 715–717, 854–855
day-count functions, 567
day-count functions in, 567
hidden header files for logical encapsulation, 763–764
hierarchical reuse of, 886–887
inappropriate physical dependencies, 742
nonprimitive functionality in, 709–714
physical dependencies, 740–744
well-factored Date class that degrades over
time, 705–714
date math, 877–878
date utilities, 881–885
date/calendar subsystem
 CacheCalendarFactory interface, 867–871
Calendar class, 895–899
calendar library, application-level use of,
862–872
CalendarCache class, 861–867
CalendarFactory interface, 867–871
CalendarLoader interface, 862–867
CurrentTimeUtil struct, 849–853
date and calendar utilities, 881–885
Date class
class design, 838–849
hierarchical reuse of, 886–887
indeterminate value in, 842
value representation in, 887–895
date math, 877–881
Date type, 838–849
DateConvertUtil struct, 889–894
DateParserUtil struct, 873–876, 895
day-count functions, colocation of
 ISMA 30/360 day-count convention, 567
PSA 30/360 day-count convention, 567
bbldc package implementation, 570–574
library date class, 567
package implementation, 575–576
protocol class implementation, 573–575
single-component implementation,
568–570
DayOfWeek enumeration, 611–613, 839
DayOfWeekUtil class, 611–612
Dealer interface, 658–660
deallocate method, 778
decentralized package creation, 421
declarations
aspect functions, 335
consistency in, 194–201
defined, 153–154, 935
definitions compared to, 154–159
forward, 358
inline functions, 778–783
local, 507, 594, 794
at package namespace scope, 312–321, 483,
938, 940
program-wide unique addresses, 163–166
pure, 188, 358
summary of, 188–190, 261–265
typedef, 168, 313
using, 328–333, 938
visibility of, 166–170
default allocators, 860
Default class, 785
DEFAULT_CHUNK_SIZE value,
785–787
defensive programming, 195
definitions
compiler access to definition source code, 166–168
declarations compared to, 154–159
declaring in header (.h) files, 212–214, 344
defined, 153–154, 935
entities requiring program-wide unique addresses, 163–166
global, 475, 762
local, 475
ODR (one-definition rule), 158, 185–186, 262–264
self-declaring, 155, 188, 261
summary of, 188–190, 261–265
visibility of, 166–170
demotion. See also levelization techniques
importance of, 95, 518–521, 941
library software, 95
overview of, 14, 461, 614–618
shared code, 436–437
summary of, 915
dependencies. See also hierarchical testability
requirement; levelization techniques; relationships
acyclic
component collections, 93–95
components, 362–370
defined, 936
levelization and, 251–256, 602
libraries, 149–151, 417–421
package groups, 411–413
package prefixes, 322–326, 937
allowed
defined, 936
title manifests and, 281–284
package groups, 408–413, 939–941
packages, 389–394, 451–454, 939–941
physical aggregates, 300, 942
compile-time, 239, 359–362
avoiding unnecessary, 778–783
defined, 936
encapsulation, 773–776
pervasiveness of, 778
real-world example, 783–789
shared enumerations, 776–777
summary of, 790, 920
cyclic. See cyclic dependencies
definitions of, 278
dependency injection, 733
dependency metadata
aggregation levels and, 473–474
implementation of, 474–475
overview of, 471–472
weak dependencies, 472–473
Depends-On relationship, 237–243
eliminating with callbacks
function callbacks, 649–651
functor callbacks, 652–654
extracting actual, 256–259, 268
implied, 220, 243–251, 267, 435
library, 146–151, 758–760
link-time
defined, 240, 936, 942
excessive dependencies, avoiding, 704–722, 916
inappropriate dependencies, 739–753, 918–919
insulation and, 802–803
local component, 447–451
modularization and, 521–523
overview of, 411–413
package
allowed, 389–394, 451–454
cyclic, 394–395
dependency metadata, 471–475
physical package structure and, 388
package-group, 408–413, 420–421, 937
physical aggregate
allowed, 281–284, 300, 942
cyclic, 292–295
definitions of, 278
dependency metadata for different levels of aggregation, 473–474
procedural interface, 813–814
test-driver, 445–447, 491–492
allowed test-driver dependencies across packages, 451–454, 940
import of local component dependencies, 447–451
minimization of test-driver dependencies on external environment, 454–456
deployment
application versus library software, 11
enterprise-wide unique names, 461
flexible software deployment, 459–460, 462–463
library software, 464
overview of, 459
package group organization during, 413–414
partitioning of deployed software, 940
business reasons, 467–469
engineering reasons, 464–467
redemption, 787
software organization, 460–462
stylistic rendering within header files, 462–463
summary of, 469, 492–493
unique .h and .o names, 460, 937
design, logical
components, 49–55
naiveté of, 497
role of, 124
design, physical. See physical design
design notation. See notation
design patterns. See patterns
destructors
documentation of, 842
Link objects, 671
protocol, 226
developer mobility, 47
development teams, autonomous core, 98–100
difference function, 566
Dijkstra, Edsger Wybe, 21
directed acyclic graph (DAG), 251–252
direction, in software design space, 498
directives
#include
component design rules, 359–362, 940
component functionality accessed via,
257–259, 346
external include guards, 205–208, 353
hierarchical testability, 447, 449, 940
internal include guards, 203–209, 353, 939
processing of, 130
removing unnecessary, 258
source-code organization, 334, 939
summary of, 265, 936
syntax and use, 201–203, 942
transitive includes, 227, 359–360, 486, 605–609, 937
using, 201, 328–333, 938
directories
doc, 388
include, 388
lib, 388
package
allowed dependencies, 389–394, 451–454, 940
physical package structure and, 388–389
disjoint clients, colocation of classes with,
524–526
DLLs (dynamically linked libraries), 153, 833
doc directory, 388
documentation
application versus library software, 10
destructors, 842
iterators, 548
type constraints, 234–236
domain independence, 756
domain-specific conditional compilation,
754–758, 941
Downey, Steve, 761
drivers, test. See test drivers
Dstack class, 774–775
dual bindage, 160–163, 263, 584–585, 935
dumb data, 629–633, 915
dummy implementations, 656, 744
dumpbin command, 133
duping, 573
dynamic programming, 70–71
dynamic storage, 162
dynamically linked libraries (DLLs), 153, 833
E
Edge objects
  dumb-data implementation, 629–633
  factoring, 675–676
  manager classes, 673–674
  opaque pointers and, 625–629
Eiffel, 33
  *The Elements of Programming* (Stepanov), 235
ellipses (.), 238
Emerson, R. W., 46
employee/manager functionality
  architectural perspective of, 618–629
  colocation, 526
  cyclic physical dependencies, 505–507
  data callbacks, 641–643
  encapsulation. See also insulation; wrappers
  compile-time dependencies, 773–776
  defined, 790–791, 920, 937
  escalating
    advantages of, 516–517, 701–703
    encapsulating wrapper, 679
    example of, 364–367
    graph subsystem example, 681–682
    history of, 688–689
    misuse of, 702
    multicomponent wrappers, 687–691
    overhead due to wrapping, 687
    package-sized systems, wrapping, 693–701
    reinterpret_cast technique, 692–693
    single-component wrapper, 685–686
    spheres of encapsulation, 679, 683
    summary of, 486, 915
    use of encapsulation components, 683–684
  insulation compared to, 791–793
  larger units of, 508
  logical, 762–765
  modules and, 475, 508
Polygon example
  “are-rotationally-similar” functionality, 541–544
  flexibility of implementation, 535–537
  implementation alternatives, 534–535
  interface, 545–552
  invariants imposed, 531
  iterator support for generic algorithms, 539–540
  nonprimitive functionality, 536–537, 541
  performance requirements, 532–533
  Perimeter and Area calculations, 537–539
  primitive functionality, 533–534, 540
  topologicalNumber function, 545
  use cases, 531–532
  values, 530
  vocabulary types, 530–531
  single-component-wrapper approach, 516
  of use, 792–793
  enterprise namespaces, 309–310
  enterprise-specific policy metadata, 476–478, 493
  enterprise-wide unique names, 461
  entity manifests, 281–283, 936
  entity/relation model, 594–596
  enum class, 313
  enumerations
    compile-time dependencies, 776–777
    component design rules, 348
    day-count example, 576
    DayOfWeek, 611–613, 839
    enum class, 313
    integral types, 576
    linkage, 170–171
    overview of, 348
  envelope/letter pattern
    aggregation of components into packages, 586–589
    event-driven programming, 576–586
    blocking functions, 576–577
    classical approach to, 577–579
    modern approach to, 579–586
    time multiplexing, 577
index of, 555, 583–586
package-level functor architecture, 586–589
equality operator (==), 221–222, 511, 882
escalating encapsulation
advantages of, 516–517, 701–703
encapsulating wrapper, 679
example of, 364–367
graph subsystem example, 681–682
history of, 688–689
misuse of, 702
multicomponent wrappers, 687–691
overhead due to wrapping, 687
package-sized systems, wrapping, 693–701
reinterpret_cast technique, 692–693
single-component wrapper, 685–686
spheres of encapsulation, 679, 683
summary of, 486, 915
use of implementation components, 683–684
Event class
  const correctness, 624
  non-const access, 624
event loops, 577
event-driven programming, 576–586
  blocking functions, 576–577
  classical approach to, 577–579
  modern approach to, 579–586
  time multiplexing, 577
Event/EventMgr subsystem, 647–648
EventQueue class, 615–618
exceptions
  exception-agnostic code, 62
  exception-safe code, 62
  procedural interfaces, 831–833
  throwing, 718–719
exchange adapters, 754–758
executables
  linking, 126, 131–132
  naming conventions, 131
  terminology for, 131
explicit keyword, 548
explicit specialization, 174–179
exposed base types, 829
extension without modification (open-closed principle), 31–40
Account report generator example, 37–40
design for stability, 43
HTTP parser example, 31–33
list component example, 33–36
malleable versus reusable software, 40–42
summary of, 117
extern keyword, 183–185, 346
external bindage, 160–163, 263, 935
external include guards, 205–208, 265, 353
external linkage, 158, 262–263, 938
externally accessible definitions, declaring in header (.h) files, 212–214, 344
extracting protocols, 799–800
extreme programming (XP), 29
facades, 573, 807–810, 830–831
factories, 505
factoring
  application versus library software, 6–13
  collaborative software, 14–20
  continuous refactoring, 14, 634
  cracked plate metaphor, 14–20
defined, 14
  hierarchical reuse, 676
    finely graduated, granular structure, 20–27, 42
    frequency of, 42
  inadequately factored subsystems, 14–20
  overview of, 14–20, 674–676
  reusable solutions and, 14–20
  toaster toothbrush metaphor, 14–20
Factory design pattern, 809–810
F.A.S.T. Group, 89, 783
f.cpp file, 159–170
feedback, 115
file1.cpp, 163–165
files
  assembly code (.s), 129
  .cap, 433
executables
  linking, 126, 131–132
  naming conventions, 131
  terminology for, 131
header (.h)
  architectural significance of, 280–281
  build process, 129–134
  in coarse-grained modular programs, 192
  declaration consistency in, 194–201
  external bindage, 214–216, 344–345
  external linkage, 212–214, 344–345
  in fine-grained modular programs, 193–194
  as first substantive line of code, 210–212, 343–344
  hiding for logical encapsulation, 762–765, 942
  macros in, 212
  modularization of logical constructs, 214
  overview of, 48, 119, 190–201
  pqrs_bar.h, 355–359
  private, 192, 279, 352
  purpose of, 128–129, 190–191
  source-code organization, 333–336, 938–939
  structs in, 9
  stylistic rendering within, 463–464
  summary of, 264–265, 937–939
  unique names, 460
  in unstructured programs, 191–192
implementation. See implementation (.cpp)
files
  names, 292
object (.o)
  atomicity of, 131–134
  build process, 131–134
  naming conventions, 131
  sections, 135, 138–139
  static initialization, 152
  undefined symbols in, 133, 146
  unique names, 460
  weak symbols in, 138–139
zero initialization, 131–132
translation units (.i), 129, 259–260, 262
file-scope static objects, runtime initialization of, 354–359, 939
fine-grained modules, components as, 498
fine-grained unit testing, 438
finely graduated, granular structure, 23–27, 31, 42, 118
fixed-size allocation, 783
flags, policy metadata, 477–478
“flea on an elephant” colocation criteria, 559–560, 591
flexible software deployment
  importance of, 459–460
  need for, 462–463
  stylistic rendering within header files, 463–464
  summary of, 492–493
Flyweight pattern, 900
focused purpose, need for, 527
Foo class, 156, 355
FooUtil class, 179–183
for syntax, 797
FormatUtil, 61
Fortran, 125
forward declarations. See pure declarations
frameworks, metaframeworks, 47
free functions, 126, 178
  scope of, 199–200, 312–321
  source-code organization, 335
free operators
  colocation of, 560
  declaring at package namespace scope, 312–321, 483, 938
  overloading, 319–320
  source-code organization, 335
friendship
  colocation and, 556–557, 591
  constraints on, 508, 939
  friend declaration, 692
fully insulating concrete wrapper component, 687
  example of, 805–807
  performance impact of, 807
poor candidates for, 807–810
usage model, 804–807
fully qualified names, 311
functions and methods
absEqual, 34
addDaysIfValid, 844
addNode, 667, 673
advanceMonth, 878–879
allocate, 699, 778
alphabetizing in sections, 845
asDateTimeTz, 849
aspect, 311, 335, 423, 483, 839, 937–938
blocking, 576–577
calculateOptimalPartition, 60, 67
callbacks
cyclic rendering of Event/EventMgr
subsystem, 647–648
defined, 643–644
disadvantages of, 651
eliminating framework dependencies with,
649–651
function callbacks in main, 644–647
compare, 172–174
consume, 699
d_freeList_p, 776, 781
deallocate, 778
destructors, 842
difference, 566
extern template, 183–185
free, 126, 178
scope of, 199–200, 312–321
source-code organization, 335
function-call syntax, 652
generateResponse, 746
getYearMonthDay, 845
inline, 511, 539, 778–783
component design rules, 354
linkage, 166–168, 171–172, 177
source-code organization, 336
substitution, 21
insertAfterLink, 328
invoke, 652
isBusinessDay, 896
isLeapYear, 839
isNonBusinessDay, 896
isValidYearMonthDay, 610, 844, 895
load, 862
loadPartition, 79
main, 126–128
function callbacks in, 644–647
multifile program example, 133–134
“singleton” registry example,
144–145
metafunctions, 564
minCost1, 79
myTurnUpTheHeatCallback function, 795
nested class constructors, 375
nthDayOfWeekInMonth, 881
numbers of, 9
numBitsSet, 898
numMonthsInRange, 877
op, 126–127
organizing in source code, 336
overloading, 174
procedural-interfaces functions, 813–814,
823–824
“raw,” 538–539
removeNode, 673
replenish, 784–789
set_lib_handler, 645–646
shiftModifiedFollowingIfValid, 883
signatures, 127
size, 781
static, 159, 161, 315–316
streamIn, 839
streamOut, 664, 839
swap, 550
template, 669, 732
explicit specialization, 175–179
properties of, 172–175
topologicalNumber, 545
turnUpTheHeat, 795
type-safe linkage, 127
virtual, 797, 803
functors
  callbacks
    defined, 651
    eliminating framework dependencies with, 652–654
    inline functions, 652–654
    stateless functors, 654–655
    defined, 579
    event-driven programming with, 579–586

G
g.cpp file, 159–170
  generateResponse function, 746
  generic algorithms, iterator support for, 539–540
  getYearMonthDay method, 845
  global definitions, 475, 762
  global resources, 762
  GMT (Greenwich Mean Time), 849
  goals, software development, 3–5, 115
  Google, 519
  grandfathering, 473
  granular software, 23–27, 31, 42, 118
  graph subsystem
    Edge objects
      dumb-data implementation, 629–633
      factoring, 675–676
      manager classes, 673–674
      opaque pointers and, 625–629
    escalating encapsulation
      history of, 688–689
      individual spheres of encapsulation, 681–682
      multicomponent wrappers, 687–691
      overhead due to wrapping, 687
      package-sized systems, wrapping, 693–701
      reinterpret_cast technique, 692–693
      single-component wrapper, 685–686
      use of implementation components, 683–684
    Node objects
      dumb-data implementation, 629–633
      factoring, 675–676
      manager classes, 673–674
      opaque pointers and, 625–629
      greedy algorithms, 59
      Greenwich Mean Time (GMT), 849
      Gregorian calendar, 610, 886
      groups, package, 942. See also library software;
        modularization
        bsl (BDE Standard Library), 404–406
        defined, 82, 271–272, 402, 937
        dependencies, 408–413, 937, 939–941
        naming conventions, 326–327, 402–403, 423–424, 937, 939
        notation, 406–408
        organizing during deployment, 413–414
        package names within, 504–505, 939
        physical aggregation with, 402–413
        practical applications, 414–421
        acyclic application libraries, 417–421
        decentralized package creation, 421
        purpose of, 414–417
        role of, 402, 942
        summary of, 421–422, 427, 488–490, 940
      GTest, 456

H
  .h files. See header (.h) files
  Halpern, Pablo, 788
  handles, 516–517
  hash table, text-partitioning optimization, 81
  header (.h) files. See also components; directives
    architectural significance of, 280–281
    build process, 129–134
    in coarse-grained modular programs, 192
    declaration consistency in, 194–201
    external bindage, 214–216, 344–345
    external linkage, 212–214, 344
    in fine-grained modular programs, 193–194
    as first substantive line of code, 210–212, 343–344
    hiding for logical encapsulation, 762–765, 942
    macros in, 212
    modularization of logical constructs, 214
    overview of, 48, 119, 190–201
pqrs_bar.h, 355–359
private, 192, 279, 352
purpose of, 128–129, 190–191
source-code organization, 333–336, 938–939
structs in, 9
stylistic rendering within, 463–464
summary of, 264–265, 937–939
unique names, 460, 937
in unstructured programs, 191–192
heavy layering, 729
“Hello World!” program, 125–126
helper classes, component-private, 561–564
heterogeneous development teams, 98–100
hidden header files for logical encapsulation, 762–765
hierarchical reuse. See also date/calendar subsystem; physical interoperability
Date class, 886–887
designing for, 10
factoring and, 676
finely graduated, granular structure, 20–27, 42
frequency of, 42
finely graduated, granular structure, 20–27, 42
frequency of, 42
hierarchical testability requirement, 437
allowed test-driver dependencies across packages, 451–454, 940
associations among components and test drivers, 441–445
black-box testing, 445
dependencies of test drivers, 445–447, 940
directory location of test drivers, 445, 940
fine-grained unit testing, 438
import of local component dependencies, 447–451
#include directives, 447, 449, 940
minimization of test-driver dependencies on external environment, 454–456
need for, 439–441, 940
summary of, 458–459, 491–492
uniform test-driver invocation interface, 456–458, 941
“user experience,” 458, 941
white-box knowledge, 445
overview of, 20–27, 676
software repository, 108–109
summary of, 117
system structure and, 20–27
text-partitioning optimization analogy, 57–86
brute-force recursive solution, 64–70
component-based decomposition, 60–64
dynamic programming solution, 70–76
exception-agnostic code, 62
exception-safe code, 62
greedy algorithm, 59
lookup speed, 79–83
nonlinear global cost function, 59
probability of reuse, 84–86
problem summary, 57–59
real-world constraints, 86
reuse in place, 76–79
summary of, 119–120
vocabulary types, 85
hierarchical testability requirement, 437
allowed test-driver dependencies across packages, 451–454, 940
associations among components and test drivers, 441–445
black-box testing, 445
dependencies of test drivers, 445–447, 940
directory location of test drivers, 445, 940
fine-grained unit testing, 438
import of local component dependencies, 447–451
#include directives, 447, 449, 940
minimization of test-driver dependencies on external environment, 454–456
need for, 439–441, 940
summary of, 458–459, 491–492
uniform test-driver invocation interface, 456–458, 941
“user experience,” 458, 941
white-box knowledge, 445
hierarchy, protocol, 231
holidays, date/calendar subsystem, 855, 859
horizontal library development, 811
horizontal packages, 414–415, 502
horizontal subsystems, 730
HTTP parser, 31–33

I
.i files, 129–130, 259–260
.i suffix, 805
“ill-formed” programs, 692–693
implementation (.cpp) files
  architectural significance of, 280–281
  build process, 129–134
  compiling and linking
    build process, 129–134
    defined, 129
    executables, 126, 131–132
    library archives, 139–141
    object files (.o), 131–139
    “singleton” registry example, 141–146
  summary of, 259–260
externally accessible definitions, 212–214, 344
f.cpp, 159–170
file1.cpp, 163–165
g.cpp, 159–170
implementation components, 677
.m.cpp suffix, 435
overview of, 48, 119, 124
partitioning, 281
source-code organization, 341–342, 938
structs in, 9
implementation-specific interfaces, 802
implied dependency, 220, 243–251, 267, 435
inadequately factored subsystems, 14–20
inappropriate link-time dependencies, avoiding
  “betting” on single technology, 745–753
inappropriate physical dependencies, 740–744
overview of, 739
summary of, 753, 918–919

#include directives, 130
component design rules, 359–362, 940
external include guards, 205–208, 353
header (.h) files, 257–259, 346
hierarchical testability, 447, 449, 940
internal include guards, 203–209, 939
  component design rules, 353
  examples of, 205
  external include guards compared to, 205–208
  need for, 203–205
  removing unnecessary, 258
source-code organization, 334, 939
summary of, 265, 936
syntax and use, 201–203, 942
transitive includes, 227, 359–360, 486, 605–609, 937
include directory, 388
independent solutions, 45
indexed lookup, 79–83
inequality operator (!=), 221–222, 511
inherently primitive functionality
  in higher-level utility structs, 529–530
  overview of, 528–529
 Polygon example
    “are-rotationally-similar” functionality, 541–544
    flexibility of implementation, 535–537
    implementation alternatives, 534–535
    interface, 545–552
    invariants imposed, 531
    iterator support for generic algorithms, 539–540
    nonprimitive functionality, 536–537, 541
    performance requirements, 532–533
    Perimeter and Area calculations, 537–539
    primitive functionality, 533–534, 540
    topologicalNumber function, 545
    use cases, 531–532
    values, 530
    vocabulary types, 530–531
    quick reference, 941
    reducing with iterators, 529, 942
inheritance
  constrained templates and, 230–233
  equivalent bridge pattern, 801
  inheritance-based lateral architectures, 732–738
  “inheritance” relationships, 234
  private, 692
  procedural interfaces, 828–829
  public, 359–362
in-house expertise, 107–108
initialization
  runtime, 354–359, 939
  static, 152
  zero initialization, 131–132
inline functions, 511, 539, 778–783, 939
  component design rules, 354
  linkage, 166–168, 171–172, 177
  source-code organization, 336
  substitution, 21
inline variables, 162
insertAfterLink function, 328
In-Structure-Only collaborative logical relationship, 227–230
insulation. See also wrappers
  defined, 790–791, 793–794, 937
  encapsulation compared to, 791–793
  fully insulating concrete wrapper component, 687, 795
  example of, 805–807
  performance impact of, 807
  poor candidates for, 807–810
  usage model, 804–807
  goals of, 791
insulated details, 279–280
modules and, 793, 811
overview of, 790, 794–795
procedural interfaces, 804–807
  architecture of, 812–813
  defined, 810–811
  DLLs (dynamically linked libraries), 833
  example of, 816–819
  exceptions, 831–833
  functions in, 813–814, 823–824
  inheritance, 828–829
mapping to lower-level components, 815
mitigating cost of, 830–831
  naming conventions, 819–823
  physical dependencies within, 813–814
  properties of, 812–813, 825–826
  return-by-value, 826–827
SOAs (service-oriented architectures), 833
  supplemental functionality in, 814
  templates, 829–830
  vocabulary types, 824–825
  when to use, 811–812
protocols
  advantages of, 795–798
  bridge pattern, 801
  effectiveness of, 802
  extracting, 799–800
  implementation-specific interfaces, 802
  runtime overhead, 803–804
  static link-time dependencies, 802–803
  summary of, 790, 834–835, 920–921
  total versus partial, 782, 793–794, 835
  virtual functions, 669
  when to use, 765
int state, 531
interfaces. See also inheritance; logical/physical name cohesion
  abstract, 498–499, 526
  Blackjack model, 658–660
  CacheCalendarFactory, 867–871
  CalendarFactory, 867–871
  CalendarLoader, 862–867
  implementation-specific, 802
  policies, 654
  Polygon example, 545–552
procedural
  architecture of, 812–813
  defined, 810–811
  DLLs (dynamically linked libraries), 833
  example of, 816–819
  exceptions, 831–833
  functions in, 813–814, 823–824
  inheritance, 828–829
  mitigating cost of, 830–831
  naming conventions, 819–823
physical dependencies within, 813–814
properties of, 812–813, 825–826
return-by-value, 826–827
SOAs (service-oriented architectures), 833
supplemental functionality in, 814
templates, 829–830
vocabulary types, 824–825
when to use, 811–812
programmatic, 390, 792
surface area, 16, 42
testability of, 49
types, 741–742
well-defined, 49
internal bindage, 160–162, 263, 805, 935
internal include guards
component design rules, 353
elements of, 205
external include guards compared to,
205–208
overview of, 203–209
summary of, 265
internal linkage, 159, 262–263
interoperability, physical
application-specific dependencies in library
components, 758–760, 941
constraints on side-by-side reuse, 760–761
domain-specific conditional compilation,
754–758, 941
global resource definitions, 762
goals of, 753–754
guarding against deliberate misuse, 761, 941
hidden header files for logical encapsulation,
762–765
nonportable software in reusable libraries,
766–769, 942
package-local (private) components,
769–772, 942
summary of, 772–773, 919
interpreters, 384–385
intuitively descriptive package names, 422–423
investment in Software Capital. See Software
Capital
invocable function objects. See functors
invocation interface, 456–458, 941
invoke method, 652
iostream, 126
iovec (“scatter/gather”) buffer structure, 505
irregular libraries, 431–432, 490
irregular packages, 301, 385–386, 404, 937
irregular UORs (units of release), 432
Is-A logical relationship
arrow notation, 219
implied dependency, 243–251
overview of, 219
isBusinessDay method, 895–896
isLeapYear method, 839
ISMA 30/360 day-count convention, 567
isNonBusinessDay method, 896
ISO (In-Structure-Only) collaborative logical
relationship, 227–230
isolated packages
dependencies, 420–421
naming conventions, 387, 425–426
physical layout of, 387
problems with, 387
istream operator, 873
isValidYearMonthDay method, 610, 844, 895
iterators
documentation of, 548
generic algorithms, support for, 539–540
inherently primitive functionality, reducing,
529, 942
purpose of, 34
type of, 35
J–K
Java, package scope in, 770
Kant, Immanuel, 319
keywords. See also commands; functions and
methods
explicit, 548
export, 183–185, 346
protected, 221
typename, 173
L

Lakos Polymorphic Memory Allocator Model, 271

lamdas, 61, 639

language, impact on design, 125–126

Large-Scale C++ Software Design (Lakos), 497, 602

lateral architecture

CCD (cumulative component dependency), 723
defined, 727–730
example of, 730–732
minimizing, 727–729
versus classical layered architecture, 723–726
construction analogy, 723

correspondingly layered architecture, 729

inheritance-based, 732–738
overview of, 499, 601, 722–723
protocols and, 802
purely compositional designs, improving, 726–727
summary of, 738–739, 909, 917–918
testing, 738

layered architectures

CCD (cumulative component dependency), 723
defined, 727–730
example of, 730–732
minimizing, 727–729
classical layered architecture, 723–726
construction analogy, 723
correspondingly layered architecture, 729
defined, 223
versus inheritance-based lateral architectures, 732–738
layered clients, 498–499
light versus heavy layering, 728–729
mail subsystem, 599
overview of, 722–723
private inheritance versus, 225, 332
protocols and, 802
purely compositional designs, improving, 726–727
summary of, 738–739, 917–918
testing, 738

leaf components, 251–253, 573–574, 936

legacy libraries, 431–432, 490

legacy subsystem, 811

letter pattern. See envelope/letter pattern

levelization techniques

callbacks
concept, 664–671
data, 640–643
function, 643–651
functor, 651–655
overview of, 639
protocol, 655–664
defined, 252
demotion
importance of, 95, 518–521
library software, 95
overview of, 14, 461, 614–618
shared code, 436–437
summary of, 915
dumb data, 629–633, 915

escalating encapsulation
advantages of, 516–517, 701–703
encapsulating wrapper, 679
example of, 364–367
graph subsystem example, 681–682
history of, 688–689
misuse of, 702
multicomponent wrappers, 687–691
overhead due to wrapping, 687
package-sized systems, wrapping, 693–701
reinterpret_cast technique, 692–693
single-component wrapper, 685–686
spheres of encapsulation, 679, 683
summary of, 486, 915
use of implementation components, 683–684

factoring
application versus library software, 6–13
collaborative software, 14–20
continuous refactoring, 14, 634
cracked plate metaphor, 14–20
defined, 14
hierarchical reuse, 20–27, 42, 676
inadequately factored subsystems, 14–20
overview of, 14–20, 674–676
reusable solutions and, 14–20
toaster toothbrush metaphor, 14–20
goals of, 602
level numbers, 251–256, 267
levelizable designs, 602
levelizable designs, defined, 936
manager class, 671–674
opaque pointers
  architectural perspective of, 618–629
cautions with, 621
defined, 254, 507
overview of, 618
protocols and, 226
restricted uses of concrete classes, 226
summary of, 915
when to use, 625
redundancy, 634–638
summary of, 602–603, 703–704, 915–916
lib archiver program, 145
lib directory, 388
library software. See also package groups;
  packages
  acyclic application libraries, 417–421
  application software compared to, 5–13
  atomicity of, 277
  Boost’s C++98, 234
  bsl (BDE Standard Library), 404–406
  calendar library, application-level use of,
    862–872
  compiling and linking, 139–141
  contracts, 9
  creating, 139–141
  defined, 6
  dependencies, 146–151, 758–760
  deployment, 464
  DLLs (dynamically linked libraries), 153, 833
global resource definitions, 762
integration with, 274
irregular, 431–432, 490, 937
legacy libraries, 431–432, 490
libreg.a, 145
linking, 139–141, 146–151, 153
nonportable software in, 766–769, 942
open-source, 433, 490
reusability of, 6–13
shared (dynamically linked) libraries, 153
std::bitset, 896
std::chrono, 895
std::list, 168
std::map, 79, 81
std::vector, 168
third-party, 431–433, 490
wrappers, 432, 436, 795
Xerces, 432
libreg.a library, 145
lifetime, software, 9
light layering, 728–729
linear test drivers, 756
Link objects, 671
link order
  build-time behavior and, 151
  runtime behavior and, 151
link phase (build process), 131–132, 260.
  See also linkage
linkage. See also declarations; definitions;
  linking
  bindage
    declaring in header (.h) files, 214–216, 344–345
    external/dual, 163, 935
    internal, 805, 935
    overview of, 160–162, 263
    class templates, 179–183
    compiler access to definition’s source code,
      166–168
    const entities, 188
    enumerations, 170–171
    explicit specialization, 174–179
    extern template functions, 183–185
external, 158, 262–263, 938
function templates, 172–179
inline functions, 166–168, 171–172, 177
internal, 159, 262–263
linkers, 131–132, 162–163, 260
logical nature of, 159
namespaces, 186–188
ODR (one-definition rule), 185–186
overview of, 153
program-wide unique addresses and,
163–166
summary of, 188–190, 261–265
type safety, 127–128
linked lists, 671–673
linkers, 131–132, 162–163, 260
linking. See also linkage
build process, 129–134
compiler programs, 136
defined, 129
executables, 126, 131–132
library software, 139–141, 146–151, 153
link order
build-time behavior and, 151
runtime behavior and, 151
link phase (build process), 131–132, 260
linkers, 131–132, 162–163, 260
object files (.o)
atomicity of, 131–134
build process, 131–134
naming conventions, 131
.o versus .obj suffix, 131
sections, 135, 138–139
weak symbols in, 138–139
zero initialization, 131–132
“singleton” registry example, 141–146
summary of, 259–260
type safety, 127
link-time dependencies
defined, 240, 936, 942
excessive dependencies, 704–705
Date class example, 705–717
physically monolithic platform adapter,
717–722
summary of, 722, 916
inappropriate dependencies
“betting” on single technology, 745–753
inappropriate physical dependencies,
740–744
overview of, 739
summary of, 753, 918–919
insulation and, 802–803
List class, 671–673
list component, 33–36
literate programming, 489
load method, 862
loadPartition function, 79
local component dependencies, testing, 447–451
local declarations, 507, 594, 794
local definitions, 475
local time, 742
locales, 855, 858–861
location. See also colocation
absolute, 500
identifying, 301–309, 501
logger facility, 599–601
logger-transport-email example
cyclic link-time dependencies, 592–601
protocol callbacks, 655–664
logical constructs
anchoring to components, 311–312,
346–353
modularization of, 214–216, 344–345
logical design. See also physical design
components, 49–55
naivete of, 497
role of, 124
logical encapsulation, hiding header files for,
762–765, 942
logical relationships
In-Structure-Only, 227–230
Is-A
arrow notation, 219
implied dependency, 243–251
overview of, 219
Uses-In-Name-Only, 226–227, 251, 618
Uses-In-The-Implementation
implied dependency, 243–251
#include directives with, 360–361
overview of, 221–225
Index

Uses-In-The-Interface
- implied dependency, 220, 243–251
- include directives with, 361–362
- overview of, 219–220

logical view components, 53–55

logical/physical coherence
- overview of, 294–297
- package groups and, 414–417
- summary of, 482–484

logical/physical name cohesion
- advantages of, 298–299
- definitions at package namespace scope, 312–321, 483, 938, 940
- design rules, 304, 938–940
- enterprise namespaces, 309–310
- goals of, 300
- history of, 298–299
- logical constructs, anchoring to components, 311–312
- macro names, 311, 483
- packages, 300–301
  - application packages, 436, 940
  - architectural significance of, 322–326
  - nomenclature, 304
  - package group names, 326–327
- point of use, identifying location from, 301–309
- summary of, 333, 482–484
- using directives/declarations, 328–333

long-distance friendship, 939

insulation and, 795

intractability resulting from, 439–441, 491. See also hierarchical testability requirement

long-term greedy, 115, 563

lookups
- ADL (argument-dependent lookup), 200, 314
- locale lookups, date/calendar subsystem, 858–861
- text-partitioning optimization problem, 79–83
- lowerCamelCase, 217, 371–372
- lowercase naming conventions
  - all-lowercase notation
    - component names, 304–305, 938
  - package group names, 423–424, 939
  - package names, 424–426, 939
  - procedural interface names, 819–820

  - component names, 304–305
  - lowerCamelCase, 217, 371–372
  - package group names, 423–424
  - package names, 424–426
  - procedural interface names, 819–820

- low-level cycles, costs of, 599

M

m_ prefix, 436

macros
- in header (.h) files, 212
- naming conventions, 311, 483
- mail subsystem, logger-transport-email example
  - cyclic link-time dependencies, 592–601
  - protocol callbacks, 655–664
- MailObserver class, 663
- main function, 126–128
  - function callbacks in, 644–647
  - multifile program example, 133–134
  - “singleton” registry example, 144–145
- malleable software, 8, 29–43
  - agile software development, 29–30
  - classical design techniques and, 30–31
  - defined, 29
  - fine-grained factoring, 31
  - manager classes and, 672–673
  - open-closed principle, 31–40
  - Account report generator example, 37–40
  - component functionality and, 40, 941
  - design for stability, 43
  - HTTP parser example, 31–33
  - iterators and, 511
  - list component example, 33–36
  - malleable versus reusable software, 40–42
  - Polygon example, 35, 530–553
- summary of, 910

sharing, 771

summary of, 117

XP (extreme programming), 29
manager class, 671–674
manager/employee functionality
architectural perspective of, 618–629
colocation, 526
cyclic physical dependencies, 505–507
data callbacks, 641–643
manifestly primitive functionality, 528–529, 942
manifests entity, 281–283, 936
mapping procedural interfaces, 815
Marshall, Thomas, 100, 469
Martin, Robert, 301
max function, 167
maximizing profit, 86
.m.cpp suffix, 435
mechanisms, 862
membership metadata, 476
memoization, 70–71
memory allocation, 808
Meredith, Alisdair, 178, 331
metadata
build requirements, 475–476, 493
“by decree,” 470
dependency
aggregation levels and, 473–474
implementation of, 474–475
overview of, 471–472
summary of, 493
weak dependencies, 472–473
membership, 476
policy, 476–478, 493
purpose of, 469–470
rendering, 478–479
summary of, 479–480, 493
metaframeworks, 47
metafunctions, 564
methods. See functions and methods
Meyer, Bertrand, 33
Meyers, Scott, 258
microsecond resolution, 852–853
MiFID regulatory requirement, 851
minCost1 function, 79
minimalism, 528, 554, 910
mnemonic naming, 298–299
mocking components, 526, 659, 733
modifiable private access, 441
modularization. See also colocation; modules
criteria for, 517–518, 942
demotion process
anticipated client usage, 523–528
failure to maintain, 518–519
importance of, 518–521
physical implementation dependencies
and, 521–523
semantics versus syntax as modularization criteria, 552–553
summary of, 553–554, 910–912
logical constructs, 214, 346–353
overview of, 517
semantics versus syntax as modularization criteria, 552–553
modules
compile-time dependencies, 778
component-private classes and, 371
goals of, 772
insulation in, 793, 811
introduction of, 283, 375, 555, 687, 722
metadata in, 475
module scope, 475
potential functionality of, 564, 693
monolithic platform adapter, 717–722
monolithic software blocks, 20–21
MonthOfYear class, 878
MonthOfYearSet type, 878–880
MonthOfYearSetUtil struct, 880
Moschetti, Buzz, 15
multicomponent wrappers, 687–691
escalating-encapsulation levelization technique, 516–517
problems with, 513–514
special access with, 515
wrapping interoperating components separately, 516
multifile program example, 133–134
multiparadigm language, C++ as, 910
multiple masters, software with, 44
multiplexing, time, 577
   See also colocation
my_ prefix, 201
mythical man month, 4, 88
The Mythical Man Month (Brooks), 4
myTurnUpTheHeatCallback function, 795

N
naivete of logical design, 497
named entities. See also naming conventions
   architectural significance of names, 292, 938
constants, 843
declarations
   aspect functions, 335
   consistency in, 194–201
   defined, 153–154
   definitions compared to, 154–159
   forward, 358–359
   inline functions, 778–783, 939
   local, 507, 594, 794
at package namespace scope, 312–321
program-wide unique addresses, 163–166
pure, 188, 358
summary of, 188–190, 261–265
typedef, 168, 313
using, 328–333
visibility of, 166–170
declarations
   compiler access to definition’s source code, 166–168
   declarations compared to, 154–159
   declaring in header (.h) files, 212–214, 344
   defined, 153–154
   entities requiring program-wide unique addresses, 163–166
   global, 475, 762
   local, 475
   ODR (one-definition rule), 158, 185–186, 262–264
   self-declaring, 155, 188, 261
   summary of, 188–190, 261–265
   visibility of, 166–170
linkage
   class templates, 179–183
   compiler access to definition’s source code, 166–168
   const entities, 188
   definition visibility, 168–170
   enumerations, 170–171
   explicit specialization, 175–179
   extern template functions, 183–185
   external, 158, 262–263, 938
   function templates, 172–179
   inline functions, 166–168, 171–172, 177
   internal, 159, 262–263
   linkers, 131–132, 260
   logical nature of, 159
   namespaces, 186–188
   ODR (one-definition rule), 185–186
   overview of, 153
   partial specialization, 179–183
   program-wide unique addresses, 163–166
   summary of, 188–190, 261–265
logical/physical coherence
   overview of, 294–297
   package groups and, 414–417
   summary of, 482–484
   overview of, 163–166
   package groups, 402–403
   program-wide unique addresses, 163–166
   qualified-name syntax, 156, 198, 264–265
   typenames, 173
namespaces
   aliases, 200
   as alternative to qualified naming, 198–201
   enterprise, 309–310
   linkage, 186–188
   namespace-scope static objects, 354–359, 939
   nonatomic nature of, 200
   package namespace scope, 312–321, 483, 938, 940
   pollution, 298
   source-code organization, 341–342, 938
naming conventions, 942. See also named entities
applications, 435–436, 940
architectural significance of names, 292, 938
base names, 292, 310, 372, 936
component names, 53, 301–309, 937–939, 942
components, 53, 937
executables, 131
logical/physical name cohesion
  advantages of, 298–299
  definitions at package namespace scope, 312–321, 483, 938, 940
design rules, 304, 938–940
enterprise namespaces, 309–310
goals of, 300
history of, 298
logical constructs, anchoring to
  components, 311–312
macro names, 311, 483
package prefixes, 304, 322–327, 436, 940
packages, definition of, 300–301
point of use, identifying location from, 301–309
summary of, 333, 482–484
using directives/declarations, 328–333
lowercase
  component names, 304–305
lowerCamelCase, 217, 371–372
package group names, 423–424
package names, 424–426
procedural interface names, 819–820
object files (.o), 131
packages
  intuitively descriptive names, weaknesses with, 422–423
  package groups, 326–327, 402–403, 423–424, 937, 939
  package names within groups, 504–505
physical design thought process, 502–503
prefixes, 201, 304, 322–326, 399–401
summary of, 427, 489–490, 942
unique names, 422–427, 937
physical entities, 218
procedural interfaces, 819–823
templates, 829–830
types, 217
unique names
  enterprise-wide, 461
  header (.h) files, 460
  object (.o) files, 460
  object files (.o), 460
  overview of, 292
  packages, 422–427
uppercase
  all-uppercase notation, 371–372, 938
nested classes
  constructors, 375
declaring, 375–377
defining, 373, 940
protected, 377
NewDelete-Allocator protocol, 860
NIH (not-invented-here) syndrome, 110
nm command, 133
Node objects, 625
  factoring, 675–676
  manager classes, 673–674
  opaque pointers and, 625–629
    dumb-data implementation, 629–633
noexcept, 808
nonlinear global cost function, 59
nonmodifiable backdoor access, 441
nonportable software in reusable libraries, 766–769, 942
nonprimitive, semantically related functionality, 501–502, 941
notation
  constrained templates
    interface inheritance and, 230–233
    type constraint documentation, 234–236
  Depends-On relationship, 218,
    237–243, 936
“inheriting” relationships, 234
In-Structure-Only collaborative logical relationship, 227–230
Is-A logical relationship, 219
arrow notation, 219
implied dependency, 243–251
overview of, 219
overview of, 216–219
package groups, 406–408
summary of, 237, 266–267
Uses-In-Name-Only collaborative logical relationship, 226–227, 251, 618
Uses-In-The-Implementation logical relationship
implied dependency, 243–251
#include directives with, 360–361
overview of, 221–225
Overview of, 219–220
Uses-In-The-Interface logical relationship
implied dependency, 220, 243–251
#include directives with, 361–362
overview of, 219–220
not-invented-here (NIH) syndrome, 110
NRVO (return-value optimization), 808
nthDayOfWeekInMonth function, 881
numBitsSet function, 898
numMonthsInRange function, 877

O
object (.o) files. See also library software; linking
atomicity of, 131–134
build process, 131–134
initialization
static, 152
zero initialization, 131
.o versus .obj suffix, 131
sections, 135, 138–139
undefined symbols in, 133, 146
unique names, 460
weak symbols in, 138–139
objects, 625. See also classes; functors; object (.o) files
allocator-aware (AA), 807–808
scope
file-scope, 354–359, 939
namespace-scope, 354–359, 939
serialization, 146
odema::Pool component, 784–789
odet::DateSequence. See DateSequence class
ODR. See one-definition rule (ODR)
OFFLINE ONLY tag, 477
Olkin, Jeffrey, 612
one-definition rule (ODR), 158, 185–186, 262–264
op function, 126–127
Opaque class, 168
opaque pointers
architectural perspective of, 618–629
cautions with, 621
defined, 254, 507
overview of, 618
protocols and, 226
summary of, 915
when to use, 625
open-source software, 271
open-closed principle
Account report generator example, 37–40
component functionality and, 40, 941
design for stability, 43
HTTP parser example, 31–33
iterators and, 511
list component example, 33–36
malleable versus reusable software, 40–42
overview of, 31–40, 528, 941
Polygon example, 35
“are-rotationally-similar” functionality, 541–544
flexibility of implementation, 535–537
implementation alternatives, 534–535
interface, 545–552
invariants imposed, 531
iterator support for generic algorithms, 539–540
nonprimitive functionality, 536–537, 541
performance requirements, 532–533
Perimeter and Area calculations, 537–539
primitive functionality, 533–534, 540
topologicalNumber function, 545
use cases, 531–532
values, 530
vocabulary types, 530–531
summary of, 117, 910
open-source libraries, 433, 490
operators
equality (==), 221–222, 511, 882
free
   colocation of, 560
declaring at package namespace scope, 312–321, 483, 938
overloading, 319–320
source-code organization, 335
inequality (!=), 221–222, 511
istream, 873
postfix, 847
relational, 846
stream-out, 819
optimization, return-value, 808
OraclePersistor class, 736
organization, software
during build process, 462
during deployment, 460–461
organizational units of deployment, package
   groups as, 413–414
OSI network model, 22
OsUtil class, 742–743
overloading
   free operators, 319–320
   functions, 174
overriding virtual functions, 797
physical aggregation with, 402–413
practical applications, 414–421
acyclic application libraries, 417–421
decentralized package creation, 421
purpose of, 414–417
role of, 402, 942
summary of, 421–422, 427, 488–490, 940
package-local (private) components, 769–772, 942
packages. See also components; library
   software; utility packages
   application, 433–437, 491, 940
   architectural significance of, 300, 322–326, 385–386
   charter, 502
   coincidental cohesion, 395–396
day-count example, 575–576
decentralized package creation, 421
defined, 300–301, 332, 384, 386, 481, 936–937
dependencies
cyclic, 394–395
dependency metadata, 471–475
physical package structure and, 388–389
factoring subsystems with, 384–394
horizontal, 414–415, 502
irregular, 301, 385–386, 404, 937
isolated
dependencies, 420–421
   naming conventions, 387, 425–426
physical layout of, 387
   problems with, 387
levelization and, 251–252
metadata
   build requirements, 475–476, 493
   “by decree,” 470
dependency, 471–475, 493
   membership, 476
   policy, 476–478, 493
   purpose of, 469–470

P
package directory, 388
package groups. See also library software
bsl (BDE Standard Library), 404–406
defined, 82, 271–272, 402, 937
dependencies, 408–413, 937, 939–941
naming conventions, 326–327, 402–403, 423–424, 937, 939
notation, 406–408
organizing during deployment, 413–414
Index

rendering, 478–479
summary of, 479–480, 493
naming conventions
intuitively descriptive names, weaknesses with, 422–423
package names within groups, 504–505, 939
physical design thought process, 502–503
prefixes, 201, 304, 322–326, 399–401
summary of, 427, 489–490, 942
unique names, 422–427, 937
notation, 388–389
package groups
bsl (BDE Standard Library), 404–406
defined, 82, 271–272, 402, 937
dependencies, 408–413
names, 326–327, 402–403
naming conventions, 326–327, 402–403, 423–424, 937, 939
notation, 406–408
organizing during deployment, 413–414
physical aggregation with, 402–413
practical applications, 414–421
purpose of, 414–417
role of, 402, 942
summary of, 421–422, 427, 488–490, 940
physical layout of, 387–388
regular, 487
scope of, 312–321, 395–399, 483, 502, 938, 940
single-threaded reference-counted functors
example
aggregation of components into packages, 586–589
event-driven programming, 576–586
overview of, 555–557
structural organization of, 270–274, 481
subpackages, 427–431, 490
suffixes, 552
summary of, 401, 487–488, 942
package-sized systems, wrapping, 693–701
PackedCalendar class, 859–861, 900–901
PackedIntArray class, 901
PackedIntArrayConstIterator type, 901
PackedIntArrayUtil struct, 901
parallel processing, 456
parentheses, 652
Parnas, D. L., 20–21
ParserImpUtil struct, 876
parsers, extension of, 31–33
partial insulation, 782, 793–794, 835
partial specialization, 179–183
partitioning
deployed software, 940
for business reasons, 467–469
for engineering reasons, 464–467
implementation (.cpp) files, 281
patches, 920
patterns
"Big Ball of Mud," 5
Factory, 809–810
Flyweight, 900
singleton, 919
peer review, 90–91
peers, 557–558
perfect competition, 87
perimeter, polygons, 537–539
persistence, date/calendar subsystem, 876–877
Persitor class, 733–738
Phonebloks, 27
physical aggregation, 940
architectural significance of, 278–281
components, 280–281
names, 292, 938
summary of, 278–280
atomicity of, 277
balance in, 284–287, 290
defined, 275, 936
dependencies
allowed, 281–284, 300, 938, 942
cyclic, 292–293
definitions of, 278, 942
dependency metadata for different levels of aggregation, 473–474
entity manifests, 281–283, 936
levels of, 287–290, 942
package groups, 402–413
physical-aggregation spectrum, 275–277
summary of, 293, 481–482
UORs (units of release)
architectural significance of, 278–280, 290–291, 942
defined, 277, 936
in isolated packages, 289
physical dependencies. See dependencies
physical design, 124. See also dependencies; encapsulation; insulation; levelization techniques; packages
class colocation
component-private classes, 561–564
criteria for, 501, 522–527, 555–560, 591, 941
day-count example, 566–576
mutual collaboration, 555–560, 941
nonprimitive functionality, 541, 941
single-threaded reference-counted functors example, 576–591
subordinate components, 564–566
summary of, 591–592, 912–914, 941
template specializations, 564
components, 54–57
date/calendar subsystem example
CacheCalendarFactory interface, 867–871
Calendar class, 895–899
calendar library, application-level use of, 862–872
CalendarCache class, 861–867
CalendarFactory interface, 867–871
CalendarLoader interface, 862–867
currentTimeUtil struct, 849–853
date and calendar utilities, 881–885
Date class, 838–849, 886–895
date math, 877–881
date type, 838–849
DateConvertUtil struct, 889–894
DateParserUtil struct, 873–876
day-count conventions, 877–878
distribution across existing aggregates, 902–907
holidays, 855, 859
multiple locale lookups, 858–861
overview of, 835
PackedCalendar class, 859–861, 900–901
ParserImpUtil struct, 876
requirements, 835–838, 854–858
summary of, 908, 922–923
value transmission and persistence, 876–877
weekend days, 855
defined, 44
importance of, 2
lateral versus layered architectures
CCD (cumulative component dependency), 727–732
classical layered architecture, 723–726
construction analogy, 723
correspondingly layered architecture, 727–732
inheritance-based lateral architectures, 732–738
light versus heavy layering, 728–729
overview of, 722–723
protocols and, 802
purely compositional designs, improving, 726–727
summary of, 738–739, 917–918
testing, 738
logical/physical coherence
overview of, 294–297
package groups and, 414–417
summary of, 482–484
logical/physical name cohesion
advantages of, 298–299
definitions at package namespace scope, 312–321, 483, 938, 940
design rules, 304, 938–940
enterprise namespaces, 309–310
goals of, 300
history of, 298
logical constructs, anchoring to
components, 311–312
macro names, 311, 483
Index

packages, 300–301, 304, 322–327, 436, 940
point of use, identifying location from, 301–309
summary of, 333, 482–484
using directives/declarations, 328–333
modularization
anticipated client usage, 523–528
criteria for, 517–518, 942
demotion process, 518–521, 552–554, 910–912
failure to maintain, 518–519
overview of, 517
physical implementation dependencies and, 521–523
semantics versus syntax as modularization criteria, 552–553
summary of, 553–554, 910–912
notation
constrained templates, 230–233
Depends-On relationship, 218, 237–243, 936
“Inheriting” relationships, 234
In-Structure-Only collaborative logical relationship, 227–230
Is-A logical relationship, 219, 243–251
overview of, 216–219
summary of, 237, 266–267
type constraint documentation, 234–236
Uses-In-Name-Only collaborative logical relationship, 226–227, 251, 618
Uses-In-The-Implementation logical relationship, 221–225, 243–251, 360–361
overview of, 496–497
physical aggregation, 940
allowed dependencies, 281–284, 300, 938, 942
atomicity of, 277
balance in, 284–287, 290
cyclic physical dependencies, 292–293
defined, 275
dependencies, 278, 281–284, 292–293, 300, 473–474, 942
entity manifests, 281–283
levels of, 287–290, 942
package groups, 402–413
physical-aggregation spectrum, 275–277
summary of, 293, 481–482
UORs (units of release), 277–280, 289–291
physical interoperability
application-specific dependencies in library components, 758–760, 941
constraints on side-by-side reuse, 760–761
domain-specific conditional compilation, 754–758, 941
global resource definitions, 762
goals of, 753–754
guarding against deliberate misuse, 761, 941
hidden header files for logical encapsulation, 762–765
nonportable software in reusable libraries, 766–769, 942
package-local (private) components, 769–772, 942
summary of, 772–773, 919
physical uniformity
developer mobility and, 47, 119. See also components
importance of, 46–47
summary of, 118–119
quick reference, 935–942
role of, 2, 44–46, 118
schedule/product/budget trade-offs, 3–5
thought processes in
abstract interfaces, 498–499
colocation, criteria for, 501, 522–527
components as fine-grained modules, 498
cyclic physical dependencies, avoidance of, 503, 505–507
direction, 498
friendship, constraints on, 508
multicomponent wrappers, 513–517
naivete of logical design, 497
nonprimitive, semantically related functionality, 501–502
open-closed principle, 511
overview of, 497
package charter, 502
package names, 502–505, 939
package prefixes, 502–504
package scope, 502
physical location, identifying, 501
private access within single component, 511
private access within wrapper component, 512–513
software reuse, 500
summary of, 517, 909–910
wrappers, 508–510
physical interoperability
application-specific dependencies in library components, 758–760, 941
constraints on side-by-side reuse, 760–761
domain-specific conditional compilation, 754–758, 941
global resource definitions, 762
goals of, 753–754
guarding against deliberate misuse, 761, 941
hidden header files for logical encapsulation, 762–765
nonportable software in reusable libraries, 766–769, 942
package-local (private) components, 769–772, 942
summary of, 772–773, 919
physical location, identifying, 501
physical name cohesion. See logical/physical name cohesion
physical substitutability, 441
physical uniformity
developer mobility and, 47, 119. See also components
importance of, 46–47
summary of, 118–119
physical view, components, 53–55
physically monolithic platform adapter, 717–722
PIMPL (Pointer-to-IMPLementation), 807
Pls. See procedural interfaces
platforms, coupling with, 741–742
Player interface, 658–660
plug-ins, 47
plus sign (+), 431–432
PMR (Polymorphic Memory Resource), 222, 785
Point class, 169–170, 816–824
point of use, identifying location from, 301–309
pointers, opaque. See opaque pointers
Pointer-to-IMPLementation (PIMPL), 807
PointList class, 239–241
policies
inappropriate physical dependencies, 742
interface, 654
policy metadata, 476–478, 493
policy-based design, 654, 744
Polygon example
“are-rotationally-similar” functionality, 541
flexibility of implementation, 535–537
implementation alternatives, 534–535
interface, 545–552
invariants imposed, 531
iterator support for generic algorithms, 539–540
nonprimitive functionality, 536–537, 541
open-closed principle, 35
performance requirements, 532–533
Perimeter and Area calculations, 537–539
primitive functionality, 533–534, 540
topologicalNumber function, 545
use cases, 531–532
values, 530
vocabulary types, 530–531
Polymorphic Memory Resource (PMR), 222, 785
polymorphic object serialization, 146
polymorphism, runtime, 415–417, 574
Pool class, 778–783
  inline methods, 781–783
  partial insulation, 782
  replenishment strategy, 784–789
population count, 898
portability, enabling, 766–769
position, absolute, 500
positions, brokerage accounts, 594
POSIX-standard proleptic Gregorian calendar, 886
postfix operators, 847
pqrs_bar.h file, 355–359
prefixes
  package, 502–504
    application packages, 436
    architectural significance of, 322–326
    my_ prefix, 201
    nomenclature, 304
    value of, 399–401
  package groups, 304, 326–327
  procedural interfaces, 823
  purpose of, 829
  z_, 815, 819–823
preprocessing phase, 129
pricing engines, 758–759
PricingModel class, 758–759
PrimitiveDateUtil utility, 894
primitiveness
  closure and, 528
  defined, 911, 937
  inherently primitive functionality
    in higher-level utility structs, 529–530
    overview of, 528–529
    Polygon example, 530–553
    reducing with iterators, 529, 942
  manifestly primitive functionality, 528–529, 942
    in Polygon example, 533–534
    quick reference, 941
private access
  within single components, 511
  within wrapper components, 512–513
private classes, 561–564
  defined, 371
  example of, 378–383
  identifier-character underscore (_), 371–377
  implementation of, 371
  modules and, 371
  summary of, 384, 486–487
private components, 769–772
private header (.h) files, 192, 279, 352
private inheritance, 692
probability of reuse, 84–86
procedural interfaces
  architecture of, 812–813
  defined, 810–811
  DLLs (dynamically linked libraries), 833
  example of, 816–819
  exceptions, 831–833
  functions in, 823–824
  inheritance, 828–829
  mitigating cost of, 830–831
  naming conventions, 819–823
  physical dependencies within, 813–814
  physical separation of PI functions, 813–814
  properties of, 812–813
  return-by-value, 826–827
  SOAs (service-oriented architectures), 833
  supplemental functionality in, 814
  templates, 829–830
  vocabulary types, 824–825
  when to use, 811–812
profit maximization, 86
programmatic interfaces, 390, 792
programs, 434. See also applications
program-wide unique addresses, 163–166
proleptic Gregorian calendar, 610, 886
proprietary software, enterprise namespaces for, 309–310
ProprietaryPersistor class, 733
protected keyword, 221
protected nested classes, 377
protocols
  Allocator, 860, 902
  b.dex StreamIn, 839
  b.dex StreamOut, 839
  cache components and, 454
  callbacks
    Blackjack model, 655–660
    logger-transport-email example, 655–660
  channel, 505
  component design rules, 352
day-count example, 573–575
  defined, 226, 936
  destructors, 226
  hierarchy, 231, 737–738
  insulation with
    advantages of, 795–798
    bridge pattern, 801
    implementation-specific interfaces, 802
    protocol effectiveness, 802
    protocol extraction, 799–800
    runtime overhead, 803–804
    static link-time dependencies, 802–803
  NewDeleteAllocator, 860
  physical position, 498–499
test implementations, 659
PSA 30/360 day-count convention, 567
pseudo package names, 498, 506
Pthreads, 768
PubGraph class, 685
public classes
  colocation of
    component-private classes, 561–564
criteria for, 501, 522–527, 555–560, 591
day-count example, 566–576
  mutual collaboration, 555–560, 941
  nonprimitive functionality, 541, 941
  single-threaded reference-counted functors
e, 576–591
  subordinate components, 564–566
  summary of, 591–592, 912–914, 941
template specializations, 564
defined, 555
public inheritance, 359–362
pure abstract interfaces. See protocols
pure declarations, 188, 358
pure functional languages, 43
purely compositional designs, improving,
  726–727
Q
qualified-name syntax, 156, 198, 264–265
quality
  schedule/product/budget trade-offs, 3–5
  of Software Capital, 110–114
quantifying hierarchical reuse, text-partitioning
  optimization analogy, 57–86
  brute-force recursive solution, 64–70
  component-based decomposition, 60–64
dynamic programming solution, 70–76
  exception-agnostic code, 62
  exception-safe code, 62
  greedy algorithm, 59
  lookup speed, 79–83
  nonlinear global cost function, 59
  probability of reuse, 84–86
  problem summary, 57–59
  real-world constraints, 86
  reuse in place, 76–79
  summary of, 119–120
  vocabulary types, 85
quick reference guide, 935–942
quotation marks ("), 202–203, 344, 369–370,
  433, 460, 490
R
race conditions, eliminating, 829
RAII (Resource Acquisition Is Initialization), 62
  “raw” methods, 538–539
realms, 599
recompilation, 773. See also compilation
Rectangle class, 604–609, 798
recursion
  brute-force text-partitioning algorithm, 68–69
  recursively adaptive development, 100–105
redeployment, 787
redundancy
  advantages of, 77
brute-force solutions based on, 668
overview of, 634–638, 916
redundant include guards,
205–209, 265
refactoring, continuous, 419, 461, 634
reference, access by, 539–540
reference-counted functors, 654
references symbol, 162
registries
  Registry class, 145
  “singleton,” 141–146
Registry class, 145
regular packages, 487
regularity in design, 353
reinterpret_cast technique, 692–693
relational operators, 846
relationships. See also dependencies
  Depends-On, 218, 237–243, 278, 936–937, 942
  implied dependency, 243–251, 267
  “inheriting” relationships, 234
  In-Structure-Only, 227–230
Is-A
  arrow notation, 219
  implied dependency, 243–251
  overview of, 219
Uses-In-Name-Only, 226–227, 251, 618
Uses-In-The-Implementation
  implied dependency, 243–251
  #include directives with, 360–361
  overview of, 221–225
Uses-In-The-Interface
  implied dependency, 220, 243–251
  #include directives with, 361–362
  overview of, 219–220
release, units of. See UORs (units of release)
relevance, software, 10
reliability, software, 9
removeNode function, 673
rendering metadata, 478–479
replenish method, 784–789
replenishment, Pool class,
784–789
report generator, extension of, 37–40
repositories, hierarchically reusable, 108–109
Resource Acquistion Is Initialization (RAII), 62
return on investment, 86–88
return-by-value, 826–827
return-value optimization (NRVO), 808
reusable software. See also date/calendar
  subsystem; demotion; hierarchical
  reuse; Software Capital
application versus library software, 5–13
classically reusable software, 18–20, 116
collaborative software, 14–20, 116
constraints on side-by-side reuse, 760–761
factoring for reuse
  application versus library software, 6–13
  collaborative software, 14–20
  continuous refactoring, 14, 634
  cracked plate metaphor, 14–20
defined, 14
  inadequately factored subsystems, 14–20
toaster toothbrush metaphor, 14–20
“fanatical obsession” with, 637–638
hiding, 769–772, 942
hierarchical reuse, 20–27. See also text-
  partitioning optimization problem
  designing for, 10
  finely graduated, granular structure,
  20–27, 42
  frequency of, 42
  software repository, 108–109
  summary of, 117
  system structure and, 20–27
  text-partitioning optimization analogy,
  57–86
malleable versus, 40–42
nonportable software in, 766–769, 942
physical design thought process, 766–769
probability of reuse, 84–86
quality in, 110–114
real-world constraints, 86
vocabulary types, 85
Rivest, Ronald, 83
rodata segment (executables), 131
root names, 302, 483, 938
RotationalIterator class, 544
rotationally similar polygons identifying, 541–544
runtime behavior, link order and, 151
runtime initialization, 354–359, 939
runtime overhead, total insulation, 803–804
runtime polymorphism, 415–417, 574

S
.s files, 129
salient attributes, 515
“sameness,” procedural interface, 825
Sankel, David, 353, 387, 436, 536, 563, 601, 612, 771
Schmidt, Douglas C., 719
scope
  components, 55–56
  free functions, 199–200
  modules, 475
  objects
    file-scope, 354–359
    namespace-scope, 354–359
  package namespace, 312–321, 483, 938, 940
  packages, 395–399, 502
  scoped allocator model, 222
SEC (Securities and Exchange Commission), 467
“security by obscurity,” 775
self-declaring definitions, 155, 188, 261
semantics
  as modularization criteria, 552–553
  value, 530, 629
serialization, 146, 665
service-oriented architectures. See SOAs
  (service-oriented architectures)
set_lib_handler function, 645–646
settlement dates, 835
shadow classes, 516–517
Shape class, 795–798
ShapePartialImp class, 799–800
ShapeType class, 808
shared enumerations, 776–777
shared libraries, 153
shiftModifiedFollowingIfValid function, 883
side-by-side reuse, constraints on, 760–761
signatures, 127
single solution colocation criteria, 557–559, 591
single technology, “betting” on, 745–753
single-component wrapper, 685–686
single-threaded reference-counted functors
  aggregation of components into packages, 586–589
  event-driven programming, 576–586
    blocking functions, 576–577
    classical approach to, 577–579
    modern approach to, 579–586
    time multiplexing, 577
  overview of, 555–556
  package-level functor architecture, 586–589
singleton pattern, 754, 919
“singleton” registry example, 141–146
size function, 781
sliders, schedule/product/budget, 4
Snyder, Van, 110
SOAs (service-oriented architectures)
  cyclic physical dependencies and, 519
  insulation and, 833
  procedural interfaces compared to, 715
Software Capital, 86–98. See also date/calendar subsystem
  advantages of, 20
  autonomous core development team, 98–100
  benefits of, 91–98
defined, 89
demotion process, 95, 941
hierarchically reusable software repository, 108–109
in-house expertise, 107–108
intrinsic properties of, 91–92
mature infrastructure for, 106–107
motivation for developing, 89–90
origin of term, 89
peer review, 90–91
quality of, 110–114
recursively adaptive development, 100–105
return on investment, 86–88
summary of, 120–121

*Software Capital* (Zarras), 89

software development. See also components;
demotion; physical design; reusable software

application software
   defined, 6
   library software compared to, 5–13
   reusability of, 6–13
   top-down design, 6–7
“Big Ball of Mud” approach, 5
bimodal, 95
changes in, 2
collaborative software, 14–20, 116
deployment
   application versus library software, 11
   enterprise-wide unique names, 461
   flexible software deployment, 459–460,
       462–464
   library software, 464
   overview of, 459
   package group organization during,
       413–414
   partitioning of deployed software,
       464–469, 940
   redeployment, 787
   software organization, 460–462
   stylistic rendering within header files,
       462–463
   summary of, 469, 492–493
   unique .h and .o names, 460
design for stability, 43
goals of, 3–5
hierarchical reuse, 10
impact of language on, 125–126
library software
   application software compared to, 5–13
   defined, 6
   reusability of, 6–13
   logical design, 124, 497
   malleability versus stability, 29–43
   agile software development, 29–30
   classical design techniques and, 30–31
   defined, 29
   fine-grained factoring, 31
   manager classes and, 672–673
   open-closed principle, 31–40
   sharing and, 771
   summary of, 117
   XP (extreme programming), 29
   NIH (not-invented-here) syndrome, 110
   policy-based, 654, 744
   quality in, 110–114, 121–122
   recursively adaptive, 100–105
   schedule/product/budget trade-offs, 3–5, 115
   Software Capital, 86–98
   autonomous core development team,
       98–100
   benefits of, 91–98
   defined, 89
   demotion process, 95, 941
   hierarchically reusable software repository,
       108–109
   in-house expertise, 107–108
   intrinsic properties of, 91–92
   mature infrastructure for, 106–107
   motivation for developing, 89–90
   origin of term, 89
   peer review, 90–91
   quality of, 110–114
   recursively adaptive development,
       100–105
   return on investment, 86–88
   summary of, 120–121
   subsystems, identification of, 11–12
   text-partitioning optimization analogy, 57–86
   brute-force recursive solution, 64–70
   component-based decomposition, 60–64
   dynamic programming solution, 70–76
   exception-agnostic code, 62
   exception-safe code, 62
   greedy algorithm, 59
   lookup speed, 79–83
   nonlinear global cost function, 59
   probability of reuse, 84–86
problem summary, 57–59
real-world constraints, 86
reuse in place, 76–79
summary of, 119–120
vocabulary types, 85
top-down, 6–7
software organization
during build process, 462
during deployment, 460–461
Sommerlad, Peter, 258
source-code organization. See also header (.h) files; implementation (.cpp) files
header (.h) files, 333–336, 938
implementation (.cpp) files, 341–342, 938
summary of, 484–485, 938
specializations
colocation of, 564
eexplicit, 174–179
partial, 179–183
spheres of encapsulation, 679, 683
stability, software, 29–43
agile software development, 29–30
application versus library software, 8–9
classical design techniques and, 30–31
defined, 29
fine-grained factoring, 31
open-closed principle, 31–40
Account report generator example, 37–40
component functionality and, 40, 941
design for stability, 43
HTTP parser example, 31–33
iterators and, 511
list component example, 33–36
malleable versus reusable software, 40–42
Polygon example, 35, 530–553
summary of, 910
summary of, 117
text-partitioning optimization problem, 76–79
XP (extreme programming), 29
Stack type, 34, 49
StackConstIterator class, 49
standard components, adoption of, 111
standard-layout types, 692
stateful allocators, 808
stateless functors, 654–655
static functions/methods, 159, 161, 315–316
static initializations, 152
static link-time dependencies, 802–803
static storage, 162
static variables, 161
std::bitset, 896
std::chrono, 895
std::list, 168
std::map, 79, 81
std::vector, 168
Stepanov, Alexander, 235–236
Stock Studio service, date/calendar subsystem
actual (extrapolated) requirements, 837–838
CacheCalendarFactory interface, 867–871
Calendar class, 895–899
calendar library, application-level use of, 862–872
calendar requirements, 854–858
CalendarCache class, 861–867
CalendarFactory interface, 867–871
CalendarLoader interface, 862–867
CurrentTimeUtil struct, 849–853
date and calendar utilities, 881–885
Date class
class design, 838–849
hierarchical reuse of, 886–887
indeterminate value in, 842
value representation in, 887–895
date math, 877–881
Date type, 838–849
DateConvertUtil struct, 889–894
DateParserUtil struct, 873–876, 895
day-count conventions, 877–878
distribution across existing aggregates, 902–907
holidays, 855, 859
multiple locale lookups, 858–861
originally stated requirements, 835–836
overview of, 835
PackedCalendar object, 859–861, 900–901
ParserImpUtil struct, 876
requirements
  actual (extrapolated), 837–838
calendar, 854–858
  originally stated, 835–836
summary of, 908, 922–923
value representation in, 887–895
value transmission and persistence, 876–877
weekend days, 855
storage
  automatic, 162
dynamic, 162
  static, 162
streamIn method, 839
streaming, BDEX, 839–848, 898, 902
streamOut method, 664, 839
stream-out operator, 819
strong symbols, 138–139
Stroustrup, Bjarne, 12, 98, 111, 236, 244,
  870–871
structs. See also classes
  as alternative to qualified naming, 198–201
BitStringUtil, 898
BitUtil, 897–898
CalendarUtil, 883
currentTimeUtil, 849–853
DateConvertUtil, 889–894
DateParserUtil, 873–876
DayOfWeekUtil, 611–612
declaring at package namespace scope,
  312–321, 483, 938
inherently primitive functionality in,
  529–530
MonthOfYearSetUtil, 880
multiple copies of, 9
PackedIntArrayUtil, 901
ParserImpUtil, 876
Point, 169–170
stylistic rendering within header files, 463–464
subordinate components, 372, 486–487,
  564–566, 591, 937, 939
subpackages, 427–431, 490
substantive use, 239
substitution, 441
subsystems. See also date/calendar subsystem;
  packages
cyclically dependent, 596–597
Event/EventMgr, 647–648
exchange adapters, 754–758
factoring with packages, 384–394
horizontal, 730
identification of, 11–12
legacy, 811
tree-like, 414–415
sufficiency, 528, 554, 910
suffixes
  component, 553
  _i, 805
  package, 552
test drivers, 441–445
util, 315, 553, 573
surface area, 16, 42
surface to volume ratio, 116
swap function, 335, 550
symbols. See also definitions
  symbol references, 162
  undefined, 133, 146
weak/strong, 138–139, 151
syntax-centric modularization criteria, 517–518
system structure
  coarsely layered architecture, 22–23
  finely graduated, granular, 23–27
  monolithic blocks, 20–21
  properties of, 21
  top-down, 25
T
  .cpp suffix, 435
TDD (test-driven development), 738–739
teams, development, 98–100
telecopying. See partitioning
templates
  extern template functions, 183–185
  function
    explicit specialization, 175–179
    properties of, 172–175
  interface inheritance and, 230–233
naming conventions, 829–830  
procedural interfaces, 829–830  
source-code organization, 335  
specializations  
colocation of, 564  
explicit, 174–179  
partial, 179–183  
template methods, 669, 732  
type constraint documentation, 234–236  
variadic, 557–558, 581, 584  
test drivers  
associating with components, 441–445, 940  
deependencies, 445–447  
allowed test-driver dependencies across packages, 451–454, 940  
import of local component dependencies, 447–451  
minimization of test-driver dependencies on external environment, 454–456  
directory location of, 445, 940  
#include directives, 447, 449, 940  
linear, 756  
overview of, 48–49  
summary of, 458–459, 491–492  
uniform test-driver invocation interface, 456–458, 941  
“user experience,” 458, 941  
white-box knowledge, 445  
testcalendarloader component, 455  
test-driven development (TDD), 738–739  
testing. See also test drivers  
hierarchical testability requirement, 437  
allowed test-driver dependencies across packages, 451–454, 940  
associations among components and test drivers, 441–445  
black-box testing, 445  
dependencies of test drivers, 445–447, 940  
directory location of test drivers, 445, 940  
fine-grained unit testing, 438  
import of local component dependencies, 447–451  
#include directives, 447, 449, 940  
minimization of test-driver dependencies on external environment, 454–456  
need for, 439–441, 940  
summary of, 458–459, 491–492  
uniform test-driver invocation interface, 456–458, 941  
“user experience,” 458, 941  
white-box knowledge, 445  
text segment (executables), 131  
text-partitioning optimization problem, 57–86  
brute-force recursive solution, 64–70  
component-based decomposition, 60–64  
dynamic programming solution, 70–76  
exception-agnostic code, 62  
exception-safe code, 62  
greedy algorithm, 59  
lookup speed, 79–83  
nonlinear global cost function, 59  
probability of reuse, 84–86  
problem summary, 57–59  
real-world constraints, 86  
reuse in place, 76–79  
summary of, 119–120  
vocabulary types, 85  
third-party libraries, 431–433, 490  
thought processes, in physical design, 497  
absolute position, 500  
abstract interfaces, 498–499  
colocation  
component-private classes, 561–564  
criteria for, 501, 522–527, 555–560, 591, 941  
day-count example, 566–576  
mutual collaboration, 555–560  
nonprimitive functionality, 541, 941  
single-threaded reference-counted functors example, 576–591  
subordinate components, 564–566  
summary of, 591–592, 912–914  
template specializations, 564
colocation, criteria for, 522–527
components as fine-grained modules, 498
cyclic physical dependencies, avoidance of, 505–507
direction, 498
friendship, constraints on, 508
multicomponent wrappers
  escalating-encapsulation levelization technique, 516–517
  problems with, 513–514
  special access with, 515
  wrapping interoperating components separately, 516
naiveté of logical design, 497
nonprimitive, semantically related functionality, 501–502
open-closed principle, 511, 910
package charter, 502
package names, 502–505, 939
package prefixes, 502–504
package scope, 502
physical location, identifying, 501
private access within single component, 511
private access within wrapper component, 512–513
quick reference, 935–942
software reuse, 500
summary of, 517, 909–910
wrappers, 508–510
thread-safe reference counting, 589
throwing exceptions, 718–719
tight coupling, 741–742
time
  multiplexing, 577
  mythical man month, 4, 88
  schedule/product/budget trade-offs, 3–5
TimeSeries class
  component/class diagram, 508–509
  hidden header files for logical encapsulation, 763–765
  wrappers, 509–510, 512–516
TimeSeriesIterator class, 508–510
toaster toothbrush metaphor, 14–20, 27–30, 116–117
top-down design, 6–7
topologicalNumber function, 545
total insulation
  defined, 793–794
  fully insulating concrete wrapper component
    example of, 805–807
    performance impact of, 807
    poor candidates for, 807–810
    usage model, 804–807
  overview of, 794–795
  procedural interfaces, 804–807
    architecture of, 812–813
    defined, 810–811
    DLLs (dynamically linked libraries), 833
    example of, 816–819
    exceptions, 831–833
    functions in, 813–814, 823–824
    inheritance, 828–829
    mapping to lower-level components, 815
    mitigating cost of, 830–831
    naming conventions, 819–821
    physical dependencies within, 813–814
    properties of, 812–813, 825–826
    return-by-value, 826–827
    SOAs (service-oriented architectures), 833
    supplemental functionality in, 814
    templates, 829–830
    vocabulary types, 824–825
    when to use, 811–812
protocols
  advantages of, 795–798
  bridge pattern, 801
effectiveness of, 802
extracting, 799–800
implementation-specific interfaces, 802
runtime overhead, 803–804
static link-time dependencies, 802–803
summary of, 834–835, 920–921
transitive closure, 259
transitive includes, 227, 359–360, 486, 605–609, 937
translation phase, 132
translation units (.i), 130, 259–260, 262
transmitting values, 876–877
transport facility, 599–600
transport subsystem, logger-transport-email example
cyclic link-time dependencies, 592–601
protocol callbacks, 655–664
tree-like subsystems, 414–415
try/catch blocks, 832
turnUpTheHeat method, 795
typedef declarations, 168, 313
typename keyword, 173
typenames, 173
types, 10, 461, 509–510, 530
ADTs (abstract data types), 192
BitArray, 895–898
in Blackjack model, 657
Calendar, 855
conforming, 172
constraints, 234–236
covariant return types, 359
Date, 838–849
DatetimeTz, 849
defined, 27, 935
envelope components, 584
exporting, 772
flexible software deployment and, 492
incomplete, 168
in insulating wrapper component, 804–805
interface, 741–742
logical/physical name cohesion and, 323–324
naming conventions, 217
PackedIntArrayConstIterator, 901
in Polygon example, 530–531
in procedural interfaces, 824–825
purpose of, 705
redundancy with, 635
safety, 127–128
specification, 229
Stack, 34
standard-layout, 692
text-partitioning optimization problem, 85
typenames, 173
when to use, 935

U
u suffix, 552
UML, 217
unconstrained attribute classes, 610
undefined behavior, 692
undefined symbols, 133, 146
underscore (_)
in component names, 53, 304,
381–383, 487, 938–939
conventional use of, 371–377
extra underscore convention, 372–377,
561, 591, 771, 939
in package names, 425
subordinate components, 381–383, 487
two-consecutive underscores, 591
uniform test-driver invocation interface,
456–458, 941
uniformity, physical, 46–57
developer mobility and, 47, 119. See also
components
importance of, 46–47
summary of, 118–119
unique addresses, 163–166
unique names
enterprise-wide, 461
header (.h) files, 460, 937
object (.o) files, 460
overview of, 292, 937
packages, 422–427
units of release. See UORs (units of release)
universal time, 742
Unix
iovec ("scatter/gather") buffer structure, 505
nm command, 133
unstructured programs, header (.h) files in,
191–192
UORs (units of release). See also package groups
architectural significance of, 278–280, 290–291, 942
defined, 277, 936
inappropriate physical dependencies, 743, 937
irregular, 432
in isolated packages, 289
mutual collaboration and, 565–566
upgrades
coerced, 32
extension without modification (open-closed principle), 31–40
Account report generator example, 37–40
design for stability, 43
HTTP parser example, 31–33
list component example, 33–36
malleable versus reusable software, 40–42, 941
summary of, 117
upercase naming conventions
all-uppercase notation, 371–372, 938
use, encapsulation of, 792–793
use of implementation components, encapsulating, 683–684
“user experience” test drivers, 458, 941
Uses-In-Name-Only collaborative logical relationship, 226–227, 251, 618
Uses-In-The-Implementation logical relationship
implied dependency, 243–251
#include directives with, 360–361
overview of, 221–225
Uses-In-The-Interface logical relationship
implied dependency, 220, 243–251
#include directives with, 361–362
overview of, 219–220
using directives/declarations, 201, 328–333, 938
UTC (Coordinated Universal Time), 849
util suffix, 315, 553, 573
utility packages, 315, 501, 910
utility structs. See also classes
BitStringUtil, 898
BitUtil, 897–898
CalendarUtil, 883
CurrentTimeUtil, 849–853
DateConvertUtil, 889–894
DateParserUtil, 873–876
DayOfWeekUtil, 611–612
MonthOfYearSetUtil, 880
multiple copies of, 9
PackedIntArrayUtil, 901
ParserImpUtil, 876

V
value types. See types
values
access by value, 532, 539–540
additive, 839
in Date class, 887–895
return by value, 826–827
semantics, 530, 629
transmitting, 876–877
value semantics, 629
value types, 530
by-value use, 168
value-preserving integrals, 176
van Winkel, JC, 4, 27, 160, 208, 519
variables
declaring at package namespace scope, 313
inline, 162
runtime initialization of, 354–359
static, 161
variadic templates, 557–558, 581, 584
Verschell, Mike, 292
vigilance, need for, 110–114, 121–122
virtual functions, 797, 803
vocabulary types. See types

W
Wainwright, Peter, 469
weak dependencies, 472–473
weak symbols, 138–139, 151
weekend days, date/calendar subsystem, 855
well-factored Date class that degrades over time, 705–714
white-box knowledge, 445
Wilson, Clay, 906
wrappers. See also encapsulation; insulation
  Basic Business Library Day Count package, 573
cyclic physical dependencies, avoidance of, 323–324
defined, 323, 512
fully insulating concrete wrapper component, 687
described, 687–691
example of, 805–807
poor candidates for, 807–810
usage model, 804–807
overhead due to, 687
private access within, 512–513
single-component, 685–686
TimeSeries example, 508–510

X-Y-Z
Xerces open-source library, 432
XP (extreme programming), 29
z_ prefix, 815, 819–823
Zarras, Dean, 89
zero initialization, 131–132
Zvector, 15