Large-Scale C++

Volume I

Process and Architecture

John Lakos

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Large-Scale C++

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

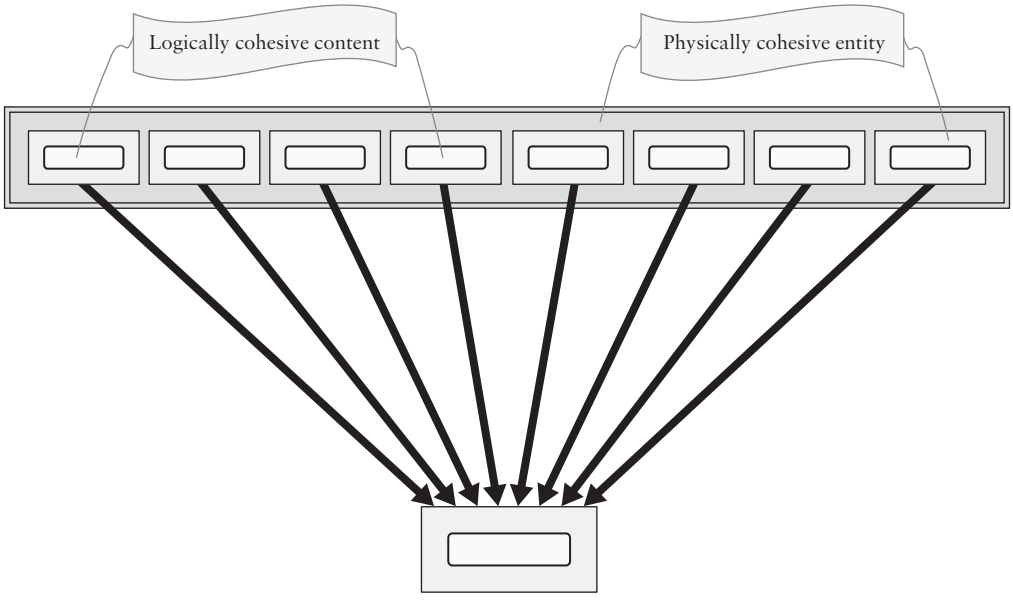

Figure 2-5: Suite of logically similar yet independent components

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

[4] See **gamma94**.

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

2.2.3 Large End of Physical-Aggregation Spectrum

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

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

2.2.4 Conceptual Atomicity of Aggregates

Guideline

Every physical aggregate should be treated atomically for design purposes.

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

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

2.2.5 Generalized Definition of Dependencies for Aggregates

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

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

Observation

The Depends-On relation among aggregates is transitive.

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

2.2.6 Architectural Significance

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

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

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

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

2.2.7 Architectural Significance for General UORs

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

2.2.8 Parts of a UOR That Are Architecturally Significant

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

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

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

---

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

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

2.2.10 A Component Is "Naturally" Architecturally Significant

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

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

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

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

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

---

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

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

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

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

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

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

2.2.14 Entity Manifest and Allowed Dependencies

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

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

Observation

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

To be practically useful, every aggregate (from a component to a UOR) must, at a minimum, somehow allow us to specify contractually the entities it aggregates, as well as the other physical