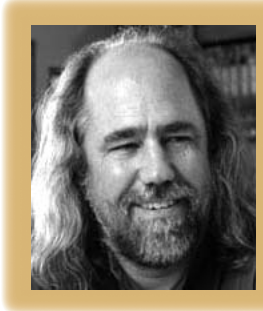
Editor: Grady Booch ■ IBM ■ architecture@booch.com

The Irrelevance of Architecture

Grady Booch

The architecture of a software-intensive system is largely irrelevant to its end users. Far more important to these stakeholders is the system's behavior, exhibited by raw, naked, running code. Actually, at the extreme, whether or not the software displays that behavior is equally irrelevant to users. As long as a system provides the right answers at the right time with all the right other "-ilities" (maintainability, dependability, changeability, and so on), end users couldn't care less about what's behind the curtain making things work.



What's behind the curtain?

Passengers consider an elevator useful if it always delivers them to their desired floors in a fair and timely fashion. Software might be behind the scenes, but it would be similarly irrelevant to the passengers if the elevator were entirely electromechanical or perhaps even run by a team of well-trained elephants. Similarly, search engines simply need to provide good-enough searches very quickly. In fact, in most cases, a perfect search that takes a really long time is far less desirable than a good-enough search that resolves quickly. This is largely because people are much better than programs at directing a search. Software is in some dimension behind the facade of every contemporary search engine, but the typical user wouldn't care if those results came instead from tens of thousands of drones sitting in a vast warehouse on a remote island—as long as the results were relevant, good enough, and fast enough.

Mind you, the operative phrase here is "at the extreme." In the case of the elevator, the more sensitive passengers would worry about the humane treatment of the elephants. In the search case, one user's single search wouldn't be a problem, but when millions of users simultaneously initiate searches, a human-intensive implementation's limits would become obvious. The point is, however, that to an end user, how a system is implemented—and thus how it's architected—doesn't really matter that much. While we software professionals obsess over the merits of C++ versus Java versus scripting languages or perhaps the obscure semantics of some corner condition of a particularly hairy protocol, users will go on with their day, happily oblivious to all our fuss and fury.

The careful reader will notice that I smudged the terms "implementation" and "architecture." I acknowledge that these are different things, but they're closely related. Specifically, a system's implementation is the manifestation of its architecture, and a system's architecture shapes its implementation. Stated another way, the architecture of a system is what we name the particular texture of a given implementation—the warp and woof of the patterns that it embodies.

When we test the behavior of a specific system implementation against a rigorous specification or, more often, an informal expectation of a system's desired behavior, we're primarily interested in gaining a high level of confidence that the functionalities of the desired system and the built system match. We want a system to do what we want it to do—to behave as expected in all normal as well as all extraordinary circumstances. For most interesting systems, unexpected circumstances will occur, and under those conditions we want the system to fail safely or behave in ways that don't astonish us. When we run tests against

 0740-7459/07/$25.00 © 2007 IEEE

a system as a whole, we're testing the particular implementation directly and its architecture only indirectly.

Who cares?

To stakeholders other than end users, however, a system's architecture is intensely interesting.

For an analyst, the presence of an emerging architecture helps bind the problem space as well as the solution space. The myriad design decisions that compose a system's architecture individually and collectively define what is and isn't a relevant part of the problem. Once thus constrained, a system's architecture provides the outline along which the analyst can explore the edges as well as the inside of a system. The evolving architecture sets the context in which the analyst can ask the system's eventual end users the right questions about desired behavior.

For the designer, a system's architecture is both a destination and a journey. As a destination, the architecture defines an ideal form to strive toward, an end point to refactor to, the visible marker of an endgame with historical precedence and a known behavior and risk profile. At any given moment, a system's architecture as designed is always somewhat unreachable. As a system's stakeholders make significant design decisions, the very activity of development and the presence of an executable architecture change the environment for the end users, analysts, and designers. This puts them in a place where they can explore issues and ask questions they simply could not have asked earlier. As a system's architecture grows, it evolves (or dies). A system's architecture as built, in contrast, provides a tangible naming of the implementation's texture in the form of architectural patterns that the designer can apply to other systems once they've proven successful.

Knowing about that texture and respecting its presence are critical to the programmers who are building a system and who, in their role as maintainers, reason about how best to evolve and adapt an existing system. Information is always lost from design to implementation, and insofar as the developers manifest those significant design decisions (that is, the architecture), current and future programmers working with that system can preserve the system's intellectual integrity as a whole.

Testers also care about a system's architecture. Most interesting system tests should be based on the use cases that are identified incrementally over the system's life cycle, the same use cases that the system's architects used to guide their design decisions. Testers can conduct other system tests only after the system's architecture is crisp. Just as analysts use a system's architecture as scaffolding along which to climb and examine the details of every edge, so too can testers use a system's architecture to devise tests that are relevant to the particular texture of that implementation.

Project managers use a system's architecture to govern its development, deployment, and evolution. A system's executable architecture becomes the primary artifact against which managers create releases, measure and manage risks, and run tests to determine an implementation's quality and the degree to which it behaves as end users desire.

Program managers also care about a system's architecture because they want to extract economies of scale from a product line. They also demand agility and resilience to change so that they can more effectively respond to and lead the changing market. Although it requires intentional effort on the part of program management, harvesting architectural patterns from successful systems globally optimizes the organization's work, giving it a competitive advantage in deploying future systems.

Modest progress

In 1994, the Standish Group published its first CHAOS Report on the state of software development. In that year, they noted that 16 percent of all software projects were successful, 31 percent were outright failures, and some 53 percent were challenged. Their most recent report, published in 2006, showed modest progress: 35 percent of all software projects were successful, 19 percent were outright failures, and 46 percent were challenged. These are still sad numbers, but at least we can claim that things are improving. The 2006 report goes on to suggest that the primary reasons for this progress were better project management, the greater use of iterative development, and leverage from the Web infrastructure.

From my world view, software architecture has a hand in all three of these factors. The best projects use a system's architecture as a primary artifact for governance. Successful projects grow a system's architecture iteratively and incrementally. The Web infrastructure is itself an architectural pattern language, which provides the structure against which a large class of interesting systems can be built.

In retrospect, I think I've titled this column incorrectly: architecture is quite relevant.

Grady Booch is an IBM Fellow and one of the Unified Modeling Language's original authors. He also developed the Booch method of software development, which he presents in *Object-Oriented Analysis and Design*. He's now working on a handbook of architectural patterns, available at www.booch.com/architecture. Contact him at architecture@booch.com.