Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

The Difference between Marketecture and Tarchitecture

Luke Hohmann

Over the years, I've been fortunate enough to help build a fairly diverse set of software solutions. Along the way, I've played various roles, including individual contributor, direct manager, and senior member of the corporate executive staff. I've worked in engineering, product marketing and product management, quality assurance, first- and second-line support organizations, and on technical publications. One thing I've learned is that a gap often exists between product management/marketing and engineering/development organizations.

Good development processes address more than how to write code.

Although we might briefly chuckle when Dilbert calls attention to this gap, most of us who've lived through it (or helped cause it) cringe at its effects: product and service offerings that ultimately fail to meet market needs. An essential step to bridging this gap is to clarify the various responsibilities associated with product management and marketing versus those associated with product engineering and development. We must also clarify the manner in which the marketing and technical aspects of the system work together to accomplish business objectives.

Who is responsible for what?

We can divide software systems architecturally along two broad dimensions. The first is the *tarchitecture* or "technical architecture" and the second is the *marketecture* or "marketing architecture." I refer to the traditional software architect or chief technologist as the tarchitect and the product-marketing manager, business manager, or program manager responsible for the system as the marketect.

The tarchitecture is the dominant frame of reference when developers think of a system's architecture. For software systems, it encompasses subsystems, interfaces, the distribution of processing responsibilities among processing elements, threading models, and so forth. In recent years, several authors, including Martin Fowler and Mary Shaw, have documented distinct tarchitecture styles or patterns, including client-server, pipeline, embedded systems, and blackboards. Our profession has begun to document how and when these various kinds of architectures are appropriate. It remains to be seen if we'll have the discipline to routinely leverage this knowledge.

Marketecture is the business perspective of the system's architecture. Among other things, it embodies

- The complete business model, including licensing and selling models
- Value propositions
- Technical details relevant to the customer
- Data sheets
- Competitive differentiation
- Brand elements
- The mental model that the marketing department attempts to create for the customer
- The system's specific business objectives

Additionally, it includes—as a necessary component for shared collaboration between the

0740-7459/03/$17.00 © 2003 IEEE

tarchitects, marketects, and developers—descriptions of functionality that are commonly included in marketing requirements documents, use cases, XP stories, and so forth.

Although tarchitecture and marketecture influence each other, marketecture dominates. Marketectural concerns—ranging from usability, installability, upgradability, supportability, deployability reliability, and other so-called nonfunctional requirements—change radically from market to market. They all influence the tarchitecture.

Examples

When I speak of the difference between marketecture and tarchitecture, most people clamor for specific examples. Here are two for your consideration.

One "heavy-client" client-server architecture that I helped create had a marketing requirement for a "modular" extension of system functionality. Its primary objective was to separately price each module and license it to customers. The business model was that, for each desired option, customers would purchase a module for the server that provided the necessary core functionality. Each client would then install a separately licensed plug-in to access this functionality. In this manner, the modules resided at both the server and client level. One example was the *extended reporting module*—a set of reports, views, and related database extract code that a customer could license for an additional fee. In terms of our pricing schedule, we sold modules as separate line items.

Instead of creating a true module on the server, we simply built all the code into the server and enabled and disabled various modules with simple Boolean flags. Those in product management were happy because the group could install and uninstall the module in a manner consistent with their goals and objectives for the overall business model. The developers were happy because building one product with Boolean flags is considerably simpler than building two products and dealing with the issues that would inevitably arise when installing, operating, maintaining, and upgrading multiple components. Internally, this approach became known as the "$50,000 Boolean flag."

The second example illustrates that the inverse to this approach can also work quite nicely. In this same system, we sold a client-side Component Object Model application programming interface that was physically created as a separate dynamic linked library. This let us easily create and distribute bug fixes, updates, and so forth. Instead of upgrading a monolithic client (which is challenging in Microsoft-based architectures), we could simply distribute a new DLL. Marketing didn't sell the API as a separate component but instead promoted it as an integrated part of the client.

The big picture

Software architecture has created various definitions. Most of the ones that I've found useful have some element of trying to capture the system's big picture. The same can be said of development processes, the majority of which try to capture development's big picture. From XP to SCRUM to "just get it done," good development processes address more issues than just how to write code. They deal with issues ranging from structuring the team to quality assurance practices to scheduling.

These are big-picture issues, but they're not big enough. If you're running a business whose products or service offerings are based on software, you need the bigger Big Picture that a comprehensive product development process can provide. When conducted properly, these processes kick in before development begins, and they are always present after development is complete. To see what I mean, watch what your development organization does after it celebrates shipping the gold master—it starts on the next version. The rest of the organization, from customer and professional services, continues to work on the release, deploying it to customers, managing upgrades, conducting press and analyst tours, and so forth.

One reason the comprehensive product development process is a bigger Big Picture is that this is where we calculate profit and loss. Put another way, most organizations view engineering as a cost center. The marketect is usually considered a "mini CEO," responsible for the product's overall financial health. The effects of this responsibility can be far-reaching, especially if they are not done effectively.

One of my clients recently asked me to conduct a business audit of one of their products. The technical team was in the process of transforming a heavy-client client-server application to a browser-based application. Everyone was thrilled with the progress they had made: customers liked the browser approach (which was easier to manage and deploy), the technical team enjoyed learning new skills, and marketing was busy making plans on how to offer the new browser-based application as an application service provider.

I wasn't so thrilled. My analysis indicated that my client was one release away from losing more than US$5 million in licensing revenue. In the old system, both the client and server were offered as a perpetual license with a 15 percent annual maintenance fee. As the development team created releases that migrated functionality from the client to the new server, my client simply shipped them to customers based on the terms of their existing maintenance agreement. Unfortunately, they had forgotten to consider that once all the functionality in the client had moved to the server, they would lose their license

My analysis indicated that my client was one release away from losing more than US$5 million in licensing revenue.

revenue associated with the client. Because they had not created any means to recoup the revenue, resolving the problem wasn't easy.

First, we delayed removing the client until we had a way to recoup the revenue. As you can imagine, this was not well received by the technical team, but once they understood the financial implications, they supported the decision. Next, I worked with the legal team to see if we had any freedom to charge upgrade fees for a new release or increase the maintenance fees on the server (because it had more functionality). We couldn't charge an upgrade fee, but we could increase maintenance fees, provided the customer hadn't locked in the maintenance agreement (several had). Finally, I suggested revamping the server's business model to separately license each browser application, increasing both list and average selling price. Although some of these changes were painful, they were less painful than losing $5 million.

What does this mean for you? Industry pundits, regardless of industry, tell us that to be successful in today's business climate, we must transcend a self-absorbed focus on our jobs and understand how our work fits into the business' larger objectives. For developers, this can mean remembering to check with your marketect department regarding the impact of a major tarchitectural change. To help you get started, send me the answer to the following questions: How does your company make money (that is, what is your business model)? What new technology will most affect your ability to lower your customer's total cost of ownership? Which nonfunctional requirement is most important to your customers? I look forward to receiving your answers.

Luke Hohmann is a management consultant for product management, software development, and organizational effectiveness. Portions of this article were based on material from his book *Beyond Software Architecture: Creating and Sustaining Winning Solutions* (Addison-Wesley, 2003). Contact him at luke@lukehohmann.com.

SOFTWARE ENGINEERING GLOSSARY

Richard H. Thayer ■ California State University, Sacramento ■ thayer@csus.edu

Software configuration management domain
(cont'd from inside back cover)

program support librarian (PSL): See *program librarian*.

release: The formal notification and distribution of an approved version of a hardware/software system. [IEEE Std. 828-1998]

software change control: The process by which a software change is proposed, evaluated, approved or rejected, scheduled, and tracked. [IEEE Std. 610.12-1990] See also *configuration management, software configuration control*.

software component (SC): A functionally or logically distinct part of a software configuration item, distinguished for the purpose of convenience in designing and specifying a complex SCI as an assembly of subordinate elements.

software configuration: 1. The arrangement of a computer system or component as defined by the number, nature, and interconnections of its constituent parts. 2. The functional and physical characteristics of a software system as set forth in technical documentation or achieved in a product. [IEEE Std. 610.12-1990]

software configuration auditing: The process of verifying that all required hardware/software configuration items have been produced, that the current version agrees with specified requirements, that the technical documentation completely and accurately describes the configuration items, and that all change requests have been resolved. [IEEE Std. 610.12-1990] See also *software configuration management*.

software configuration control: The evaluation, coordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification. [IEEE Std. 610.12-1990] See also *software configuration management*.

software configuration identification: 1. An element of configuration management consisting of selecting the configuration items for a software system and recording their functional and physical characteristics in technical documentation. 2. The current approved technical documentation for a software configuration item as set forth in specifications, drawings, associated lists, and documents referenced therein. [IEEE Std. 610.12-1990] See also *software configuration management*.

software configuration item (SCI): A software entity that has been established as a configuration item. The SCI exists where functional allocations have been made that clearly distinguish equipment functions from software functions and where the software has been established as a configurable item. Contrast with *hardware configuration item*.

—Continued on p. 56