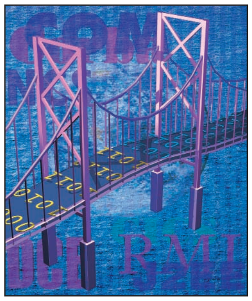

Do You Know Where Your Architecture Is?

Steve Vinoski • *IONA Technologies* • *vinoski@ieee.org*

Until June, and for almost the past five years, I was IONA's chief architect. On my first day, the head of human resources walked me around the office and introduced me as "our new senior architect." Later that afternoon, a new colleague I had met earlier approached me with some confusion about my role. "After all," he said as he looked at the walls and the ceiling, "the building looks fine. What are you planning to do to it?"

Life as a chief software architect is rarely fun. My job was to try to make sure the software underlying our products was flexible, fast, extensible, robust, consistent, cohesive, current, and devoid of duplication — and that it incorporated sound development practices. Sounds pretty obvious, as all software developers, development managers, and product managers strive for these qualities, right? Unfortunately, our old friend, the not-invented-here (NIH) syndrome, runs rampant throughout our industry. Sometimes, even when you put an architecture in place, managers and developers can still find ways to ignore it. I wish I had a dollar for every time I heard a variation of, "I don't have time to make this software conform to our product architecture; I have to get it out the door now!" Such NIH can be the result of ignorance, hubris, or outright defiance, but whatever the cause, the end result is the same: nonexistent or incoherent software architecture.

Confusion over architecture, while not uncommon in software development in general, seems prevalent in middleware. I attribute this to the fact that middleware systems are typically distributed and heterogeneous. In general, distributed systems are difficult to design, implement, debug, and maintain. When you mix in multiples of hardware platforms, operating systems, protocols, applications, and vendors, the complexity can rise to the point where nobody really understands the whole system. When systems span large enterprises and comprise multiple technologies and approaches, their keepers tend to have far more to worry about than the purity of the system's architecture.

Technology as Architecture

I recently visited a customer who told me that his company's architecture was the Java Message Service (JMS). Note that he didn't say that his company's architecture was messaging-based, or that its implementation was JMS-based; he just flat-out said that JMS is its architecture.

I've heard many similar statements over the years, and they never cease to amaze me. JMS itself is an interface specification — a Java standard for accessing and using messaging-oriented middleware (MOM) (see http://java.sun.com/products/jms/). As an interface standard, it sits firmly in the design realm. JMS can help realize an architecture, but by itself, it isn't one. Additionally, because JMS specifies the interface, not the implementation, there's no guarantee that two different JMS implementations will interoperate. The customer's statement was thus even more misguided than it first appeared: what he really implied was that his company's architecture was not just JMS, but was actually a specific vendor's JMS implementation.

Treating technology as architecture just asks for trouble. Like all technologies and specifications, JMS eventually will fall out of favor, which will make this customer's "architecture" need an upgrade. But given the customer's penchant for technology, he is likely to switch to something new well before JMS heads to the middleware retirement home. Regardless of when he switches, the cost will not be trivial. In this particular case, the customer probably used proprietary features and interfaces of the chosen JMS product. These proprietary features are likely to be intermingled with standard JMS features in the source code. Thus, even if the new technology vendor were to provide some sort of tool to help convert the legacy JMS code to the new system, the tool's utility would be limited. By failing to implement a system architecture, the customer is looking at a system rewrite with every move to a new technology.

Given the cost of redesigning a system whenever the underlying technology changes, I wonder why designers paint themselves into such corners. Conspiracy theory might promote the idea that they do it for job security, but most people I've met simply don't think that way. A lot of it is due to a simple lack of abstraction. Some designers can't seem to disassociate the problems far enough away from the technologies they use to solve them. Others don't feel they have the time to work out what the appropriate abstractions should be. Indeed, the "Get out of my way, I have to get it done yesterday" attitude that I mentioned earlier pervades the IT industry. Did some ancient middleware prophet, glimpsing a future full of brittle middleware systems, coin the phrase "haste makes waste"?

Architecture specifies not only what the system is, but also what it isn't. The more an architecture consists of abstractions, rules, and constraints that dictate what the system allows, while avoiding specific "technologies du jour," the better the chance that it will age gracefully. I've helped create middleware architecture definitions that have evolved gracefully, for example, using two simple practices. The first is to write detailed definitions for important system abstractions, or metaphors, and make sure that all team members are familiar with them. The second is to write key internal system interfaces in Corba IDL.

Writing at the abstraction level IDL affords (rather than directly in Java or C++) lets me reuse interfaces across languages and express key service interface aspects without bogging down in implementation details. Both practices let any developer with a Web browser and a chosen editor (preferably emacs, of course!) comprehend the architecture's key elements and rules at a reasonably high level of abstraction.

Poor abstraction skills can be particularly troubling in the context of service-oriented architectures.[1] Developing an SOA requires that you first identify service abstractions, but this reverses the typical approach of writing applications first and then writing the specific services that application needs. With an SOA, the goal is to put appropriately abstracted services in place for reuse by numerous applications, rather than tightly coupling the services to a single application.

By failing to implement a system architecture, the customer is looking at a system rewrite with every move to a new technology.

Coding versus Designing

During my tenure as chief architect, my employer adopted extreme programming.[2,3] While some developers were keen for XP's adoption, we were ultimately led to it when the CEO, a developer in a previous life, demanded it. Given that much of XP centers on practices similar to the highly successful iterative development methods I had introduced to the company several years earlier, I welcomed the CEO's directive.

XP, and other agile software development approaches that have followed it (see www.agilealliance.org/home/), promote iterative development with a heavy focus on continuous code review, system and unit testing, and customer interaction. The reasoning behind these approaches is that software requirements are usually very fluid. This fluidity means that trying to draft requirements before writing any code only delays the code modifications that arise from the inevitable changes in those requirements, thus prolonging the project and ultimately raising its cost. Unlike traditional waterfall-oriented approaches, agile development methods acknowledge and embrace change, rather than trying to control or prevent it.

Soon after we adopted XP, a backlash arose from various camps. The most vocal opponents were several developers with strong design and abstraction capabilities. They complained that XP threw architecture and design out the window and replaced it with "cowboy programming." They said that XP allowed developers the freedom to write whatever they wanted while providing them with a "get out of jail free" card — also known as *refactoring*[4] — should the resulting system not work as expected. Refactoring is intended to be a methodical way of improving software design and implementation without adversely affecting its external interface or usage, but it's unfortunately often simply taken as an excuse to rewrite and reinvent. While XP certainly doesn't advocate such nonsense, it can facilitate the "lone gunslinger" mentality, especially without strong communication among teammates. XP and agile methods count on each team member knowing what the other is doing; in fact, the XP practice of pair programming, where two developers share a single keyboard and interactively take turns implementing a design and its tests, is designed in part to guard against the negative effects of the cowboy programmer.

The camps that resented our adoption of XP might have preferred if we had adopted the model-driven architecture approach.[5] MDA, which follows in the traditions of structured programming and object-oriented programming, favors up-front design. Unlike these approaches, however, MDA treats code as something that tools generate from design models, rather than something that humans write. The arguments that advocates make for MDA sound logical enough. They point out that over the history of computing, we've advanced from writing machine code, to writing assembly language, to eventually writing in high-level languages such as Java and C++. With each successive step, the level of abstraction has also increased. According to their argument, the natural progression is away from low-level programming, where you explicitly write code, toward high-level development through creating models of the system, leaving the "programming" to code-generation tools. These efforts include the use of analysis patterns[6] and metadata to create a platform-independent model (PIM), from which the tools generate code — called a platform-specific model (PSM) — for a specific platform or middleware system (such as J2EE, Corba, or MOM). The PSM not only makes use of patterns specific to the underlying platform, but uses general design patterns as well.[7] In some ways, MDA requires tools that do for software what some of the tools I used earlier in my career did for defining and designing modular integrated circuits.

MDA's reliance on tools (which on the surface seems like a positive feature) might have deleterious practical effects. Middleware users know that standards help minimize lock-in to proprietary software and switching costs between standards-conforming products. Used correctly, standards also let organizations formed by mergers or acquisitions swiftly integrate their software systems, rather than having to scrap certain systems in their entirety and replace them. Unfortunately, the tools that MDA requires easily could shift vendor lock-in and ubiquity requirements from the middleware itself to the tools that create the middleware. It's true that tool standards under development might prevent this problem but, generally, it takes time for such standards to mature to the point of letting different vendors' tools interoperate cleanly.

XP and MDA advocates might not agree, but one thing the two approaches have in common is the idea of a *metaphor*. In XP terms, a metaphor is a word or short phrase that captures a project's central idea. XP metaphors are loosely similar to the names of the key entities in an MDA model. XP might use words to express metaphors whereas MDA uses modeling languages and diagrams, but the intentions are identical: both want to communicate the system architecture's key elements as succinctly, yet meaningfully, as possible.

Personally, I lean more toward XP than MDA. XP revolves directly around the main asset that makes up a living, evolving software system: the code itself. Without the code, there is no system. In my experience, MDA advocates tend to treat the code as an afterthought that tools simply generate once the appropriate models have been developed. While I believe that someday we'll realize this goal of moving to that higher level of abstraction, I think it will be a good while before we get there. Mapping from one level of abstraction to another can be difficult in practice.[8] Practical MDA advocates admit that it will be years before it is realistic to code-generate complex middleware applications that properly handle complicated issues such as multithreading, transactions, load balancing, and automatic failover. We have a lot of systems to develop before that day comes, however, and until MDA is ready for prime time, I'll stick with the iterations, shared code, close customer involvement, and refactoring practices that agile methods like XP promote.

Talk It Through

The two practices I described earlier — clearly defining system metaphors and defining interfaces or contracts using an abstract language like IDL — fit both XP and MDA. Of course your mileage might vary, but I found that when open and active communication among team members surrounds these practices, this approach easily avoids the need for wordy architecture documents or specialized tools to draw and store Unified Modeling Language diagrams.

As of mid-2003, I am no longer a chief architect. My new position is chief engineer of product innovation. As the title implies, my new job is to innovate. Such innovation might mean changes to existing products, or my work might result in new products — perhaps based on whole new architectures. Hmmm. Perhaps this means that I finally get to sit on the other side of the NIH fence for a change?

Acknowledgments

Thanks to Doug Lea and Don O'Brien for reviewing drafts of this column.

References

1. S. Vinoski, "Service Discovery 101," *IEEE Internet Computing*, vol. 7, no. 1, 2003, pp. 69–71.
2. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley Longman, 1999.
3. C. Poole and J.W. Huisman, "Using Extreme Programming in a Maintenance Environment," *IEEE Software*, vol. 18, no. 6, 2001, pp. 42–50.
4. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, 1999.
5. D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.
6. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
7. E. Gamma et al., *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
8. S.Vinoski, "It's Just a Mapping Problem," *IEEE Internet Computing*, vol. 7, no. 3, 2003, pp. 88–90.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for 15 years. Vinoski is the coauthor of *Advanced Corba Programming with C++* (Addison Wesley Longman, 1999), and he has helped develop middleware standards for the OMG and W3C. Contact him at vinoski@ieee.org.