By Matthew R. McBride

THE SOFTWARE ARCHITECT

Leadership is the defining characteristic in an unforgiving technology arena.

Adding the word architect to a software practitioner's title seems simple enough, but beneath the surface fundamentally different thinking, toolsets, and disciplines are required to succeed. In teaching software architecture and working as a software architect, database architect, and chief architect, I have often found that an unfortunate lack of knowledge surrounds the architect's role. Even experienced software practitioners are often unable to define what exactly the architect does or adds to the software development process.

The context for my discussion here is the construction of enterprise-level business applications. I chose it for its inherent difficulty and complexity, though the related architectural principles may be applied to any type of software construction. Specifically, I examine the results of applying these principles to three separate development efforts: a product-ordering Web site, a complex business-to-business integration project, and the design and development of an enterprise application. Regardless of the situation, my experience has been that software architects are not born but trained, sometimes in the school of hard knocks. Although effective software architects seem to intuitively understand and guide projects, an intellectual framework and its associated disciplines and tools are behind this thinking. Reports of IT overspending and project failure emphasize the fact that these skills must be developed. Software professionals in a variety of roles can leverage them to lead software projects to exceed customer expectations.

Many seminal ideas in software architecture can be traced back to a speech Christopher Alexander, a distinguished building architect,

delivered at the OOPSLA conference in 1996 [1, 2]. The practical application of this growing body of knowledge will continue to play an important role in the maturing of the software development profession and its ability to deliver solutions.

In my own early days as a chief architect, I encountered a business owner who demanded quarterly releases of software without regard to the system's scope or complexity. When I asked him why, he said, "Because that is the only way I will get anything from your guys." As we applied the principles of architecture, he eventually became one of my software development group's strongest allies. Another business owner who actually screamed at me regarding previously unfulfilled promises for her software product—the product-ordering Web site—became one of the group's most vocal supporters. Navigating these situations involves more than just coding skill. Fundamentally different ways of thinking about design and interacting with systems and stakeholders represent the essence of the software architect.

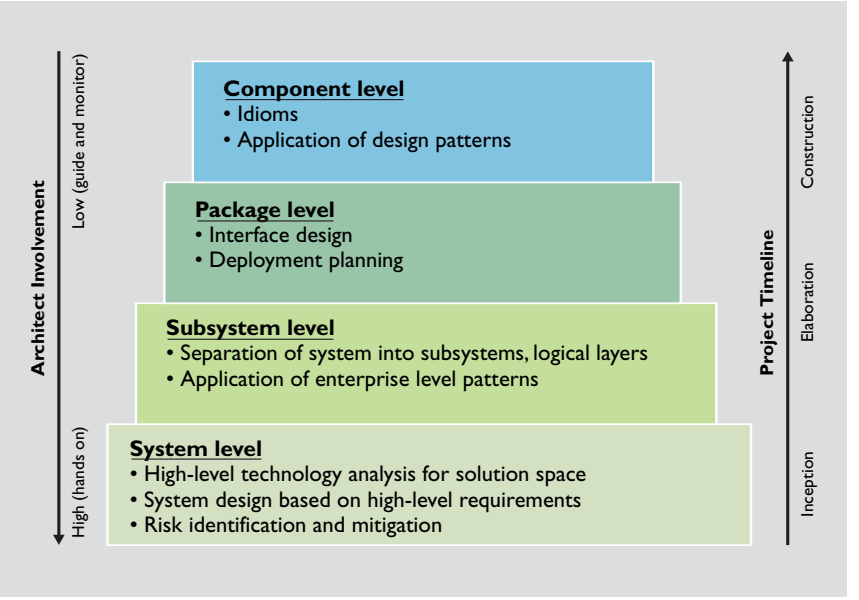

Figure 1. Architecture is the foundation of successful design.

As a starting point for this discussion, the Unified Modeling Language and other industry standards agree: architecture is a system's organizational structure [10, 11]. Some organizations allow that structure to evolve unintentionally or through neglect; others focus on designing or deriving it by following a planned process. However, allowing systems to evolve haphazardly often results in failure. Martin Fowler, a noted software development author, wrote: "In its common usage, evolutionary design is a disaster. The design ends up being the aggregation of a bunch of ad-hoc tactical decisions, each of which makes the code harder to alter. As design deteriorates, so does your ability to make changes effectively; over time the design gets worse and worse" [6]. Unfortunately, many businesses fail to consider the ramifications of poorly designed systems and suffer significant losses in terms of competitive advantage, time to market, and total cost of ownership.

Software architects are sometimes viewed by customers and developers alike as technical experts in a specific set of development technologies. Given the types of decisions they must make and influence, this perception is not surprising. However, identifying a single trait of software architects does not begin to capture the depth and breadth of their work. It is essential to proactively focus on system- and subsystem-level issues to establish a solid foundation for detailed design, particularly for large-scale efforts (see Figure 1). Software architects are technically competent system-level thinkers, guiding planned and economically efficient design processes to bring a system into existence. To do this, they must lead multiple stakeholders in a technologically challenging and sometimes politically charged environment.

The guiding principles behind the architectural decisions explored here represent an intellectual framework for any architect and a basis for success; they also represent a concrete agenda for training future architects.

Mitigate Unbounded Complexity

Almost 2,500 years ago, the Chinese philosopher and military general Sun Tzu, wrote, "If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself and not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy or yourself, you will succumb in every battle" [4]. Unbounded complexity represents a formidable enemy for any software architect. Architects must deal with the inherent complexity of both the problem and the solution domains. This complexity manifests itself [8] in two ways:

- For every 25% increase in problem (domain) complexity, there is a 100% increase in complexity of the software solution; and

- Explicit requirements explode by a factor of 50 or more into implicit (design) requirements as a software solution proceeds.

Perhaps more than any other task, managing complexity is an essential element of architecture the architect must address in order to deliver the promised system. Strategic approaches are high-level, broad-brush techniques used by architects to master complexity across technical and nontechnical audiences alike (see Figure 2). Included are effective communication and information gathering, detailed planning, and the education of all stakeholders regarding all relevant technologies. Tactical approaches address the techniques the architect employs at a lower level of detail, typically with those who construct or use the system. This group includes software designers, project teams, and end users. Requirements planning, separation of the system into logical layers, and careful interface definition are only a few of the tactical tools at the architect's disposal.

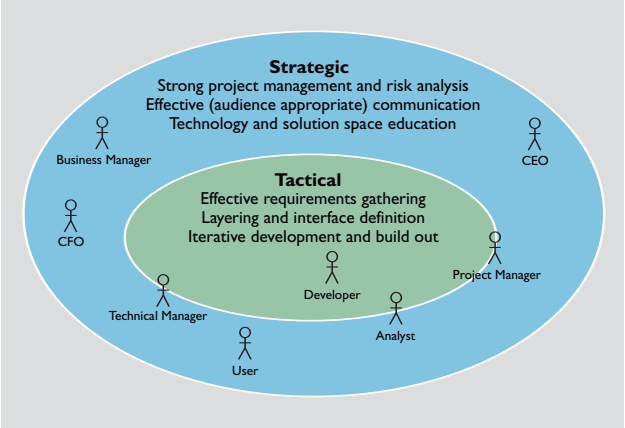

Figure 2. Multiple approaches are needed to mitigate complexity.

Logical layering and careful interface definition improve the overall design effort in several ways. First, there is a clear separation of design concerns that must be enforced. Additionally, smaller design subteams tend to form organically within boundaries provided by the logical layers, greatly reducing the complexity each design team must address. A GUI designer does not have to address the insertion of data into a database, only how it is represented to the user.

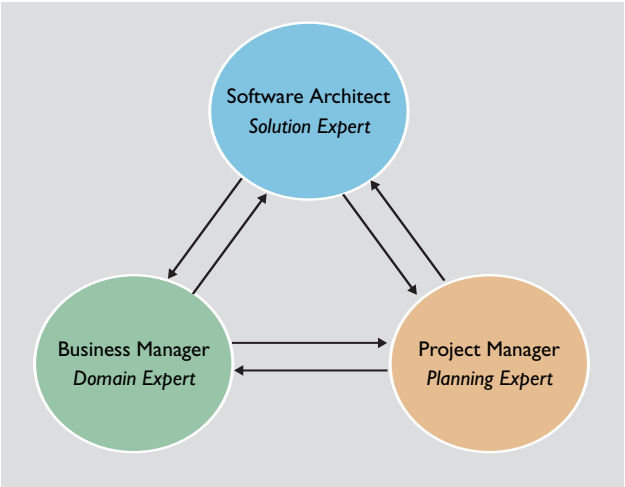

Figure 3. Sample team relationships required to build an effective solution.

Effective object-oriented code is modular; that is, it is packaged efficiently, and each component is well-defined and constructed. The architect must use similar techniques at the system level. The definition and packaging of components, subsystems, and a system's logical layers are critical to the system's performance. The system-level solution environment is not readily mastered, and without strong supervision from the software architect, projects and attempted solutions tend to fall apart due to the weight of unmitigated complexity.

Manage Functional Requirements

At the start of a project, the software architect monitors the requirements-elicitation effort to derive the high-level design solution. While architects are not necessarily planning or domain experts, they must be able to act on and process this information effectively (see Figure 3). The architect establishes several important baselines during initial requirements gathering. First, and possibly most important, is an initial understanding of the problem domain. High-level requirements and unstated expectations for the design must also be identified and validated.

During the initial requirements process, the architect must be able to perform some out-of-the-box thinking. This is important because business and domain experts understand their business and its problems but are inherently constrained by them. As the solution expert, the architect must transcend these limitations and imagine what is possible, given time and budget constraints. New approaches to old problems may have to be considered and requirements altered, added, or deleted to deliver an optimal solution.

System requirements are sometimes relatively static and unchanging. Most of the time, they represent only a first look at the problem domain, which may still be evolving. Customer expectations (both stated and unstated) and technology are also constantly changing. Perhaps the most successful way to manage this change is to deliver the software solution in iterations. Each one takes four to six weeks and results in

Without strong supervision from the software architect, projects and attempted solutions tend to fall apart due to the weight of unmitigated complexity.

deliverables (an artifact or software build that demonstrates specific functionality) the architect can discuss with the customer. Early iterations must address high-risk portions of the system design and identify key features. A greater understanding of the requirements, system-level design, and customer's priorities emerges as the process moves ahead. As customers become familiar with the emerging system, they gain confidence in a productive and responsive development team.

Communicate Effectively

The stakeholders the architect deals with vary greatly in terms of domain knowledge and software expertise and can be sorted into several categories:

Developer. Developers tend to be wonderfully creative people, though their creativity must be defined and bounded within the context of the problem domain. Moreover, their focus may at times be limited to the particular technologies and tools they use rather than to a holistic view of the solution under construction. At times, the architect must challenge this limited view to maintain the integrity of the system-level solution.

Senior-level manager and CxO-level executive. IT and non-IT executives have different needs that tend to be specific to their positions and backgrounds. Communicating with them requires the software architect's creativity and adaptation of a single vision to a specific stakeholder. For example, a CFO may tend to look at a project primarily in terms of cost and ROI to the organization. The architect must deal with the CFO on this basis, translating into financial terms the business value of a particular effort. Saying "We're using .NET" may have little meaning to a CFO, but saying "We found a way to save $20,000 and get to market two weeks earlier" will be music to this executive's ears.

Project manager. Project managers are often an architect's ally, aiming to bring order and predictability to the creative process of software construction. However, they must successfully negotiate a balance between creativity (which tends to be increasingly chaotic) and order (which limits creativity and favors predictability). In most cases, the architect allows designers and their design teams to exercise their creativity while being shepherded within reasonable schedule boundaries provided by the project manager.

Customer. Customers tend to add complexity in several ways: not clearly elaborating requirements; demanding feature-rich solutions in an unreasonable timeframe; continuously introducing new requirements; and simply failing to partner with the software design team. Nurturing a trusted partnership built on demonstrated performance by the development team can help address and mitigate these concerns.

The architect must be a translator during software construction so each stakeholder stays involved and consistently supports the proposed software solution. A number of abstractions or views may be necessary to communicate this vision across a diverse constituency. As it is communicated and validated, the architect must also identify and address the concerns of silent or unsupportive customers. The effective architect leads the diverse group of stakeholders with a servant's heart.

Embrace Leadership

The architect is the author of the solution, undeniably accountable for the effort's success or failure. While software architects may be concerned about component-level design detail, they must also champion the system-level design effort. Frederick Brooks of the University of North Carolina at Chapel Hill captured this dichotomy when discussing the need for conceptual integrity: "One must also learn a whole lore of how the [design] elements are combined in practice. Simplicity and straightforwardness proceed from conceptual integrity. Conceptual integrity in turn dictates that the design must proceed from one mind or from a very small number of agreeing resonant minds" [3].

For an architect, leadership includes the ability to provide system-level design and technical direction, work with a variety of teams and individuals, and recognize when and how to make decisions that guide the team to a successful solution.

The architect is often required to make system-

level design decisions with incomplete information early in the design process. This is risky at best, requiring a significant amount of experience and understanding of both the problem domain and the technologies involved in the solution. Competent software architects always seek the counsel of project and domain managers, technical leads, individual designers, and key customers. The decision-making process involves constantly gathering and mentally assembling relevant information.

Finally, leadership is not the same as management; it involves a variety of skills, including active concern for team members and the ability to coach, influence, and inspire. I like the way former U.S. Secretary of State Colin Powell said it: "At best (organization charts and titles) advertise some authority—an official status conferring the ability to give orders and induce obedience. But titles mean little in terms of real power, which is the ability to influence and inspire. Have you ever noticed that people will personally commit to certain individuals who on paper (or on the org chart) possess little authority—but do possess pizzazz, drive, expertise, and genuine caring for teammates and products? On the flip side, nonleaders in management may be formally anointed with all the perks and frills associated with high positions, but they have little influence on others, apart from their ability to extract minimal compliance to minimal standards" [9].

Pay Attention to Nonfunctional Requirements

Nonfunctional requirements are observable characteristics of the system as a whole. While functional requirements are primarily concerned with the stated needs of the problem domain, nonfunctional requirements represent capabilities that are orthogonal to domain requirements. They crosscut each layer of the design, as well as each design team. All too often, however, they surface as a customer's unstated expectations.

Part of the architect's role is to elicit these expectations during the initial requirements-gathering process. The architect must then identify and allocate additional nonfunctional requirements to each design subteam—a difficult and sometimes painstaking process requiring thoughtful judgment. For example, consider a three-layer system consisting of user interface, business-logic layer, and database layer. What portion of a customer's response time requirement should the user interface design team be required to meet? Each allocation and interaction must be mapped out and validated through specific system-level tests.

Many definitions of nonfunctional requirements have been proposed; the core ones by which any system is measured include availability, throughput, security, and scalability. The architect is accountable for ensuring that system performance meets user expectations. Unfortunately, if the architect does not actively address them during the initial system-design process, they could be forgotten or ignored until significant rework must be performed to address them.

Bring a Well-Stocked Toolkit

Effective software architects have a bag of tricks and tools that are largely experience-based and form the intellectual framework they use to guide decisions on a day-to-day basis. New architects may learn of them by consulting key works or references and by actively devoting time, effort, and thought to building their own. Such tools fall into several categories:

Patterns and idioms. A pattern describes a recurring problem and the core of a solution that can be modified or extended. Patterns are not specific to a particular language; rather, they seek to state the problem and outline the solution through pseudocode or plain text. Patterns provide some of the basic building blocks for both system- and component-level solutions.

The 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* formalized 23 standard design patterns [7]. Such patterns contain four essential elements: a name, a problem description, a solution, and the consequences of applying the pattern. At a component level, they provide both a common language for communication and proven outlines of solutions that may be modified and reused again and again. Several enterprise- and system-level design patterns have also been identified and put to use. They are equally useful to the seasoned architect, providing an invaluable resource for improving designs. Martin Fowler delivered a solid collection in his 2002 book *Patterns of Enterprise Application Architecture* [5].

If patterns are language-independent, idioms are their language-specific cousins. Idioms are examples of specific software language usage and conventions that represent a proven way of accomplishing a specific task. Almost every language involves generally accepted ways of connecting with a database, reading or inserting data, and closing the connection. During code reviews by system developers, compliance with these idioms must be enforced for the construction of a reliable system. Additionally, component-level development should be accomplished through a standard coding convention to promote reuse, testability, and maintainability, in addition to dramatically sim-

The architect is the author of the solution, undeniably accountable for the effort's success or failure.

plifying the work of developers and testers.

Frameworks. Several language-specific frameworks are also available, with notable contributions from Java (J2EE) and Microsoft (.NET) and associated open source and vendor toolsets. Tasks that are greatly simplified in these frameworks include concurrency, database connection pooling, and transaction management. The effective architect leverages them whenever appropriate while keeping a watchful eye on potential vendor lock-in problems. However, they often represent a double-edged sword, offering tremendous power and capability while introducing significant constraints. A prudent approach is often to delay framework-selection decisions until after the domain concepts are well formed (but not completely defined).

Best practices. Beyond the best practices identified earlier, others (such as iterative development, proactive requirements planning, test-infected development, and product-line development) can contribute significantly to a project's success. Many other disciplines (notably systems engineering) have made solid contributions along these lines and should be leveraged. Continuous learning and skill development is a hallmark of an effective software architect. The learning process restocks the toolkit, allowing the architect to bring the right tool to bear during each phase of a system's design.

Reporting Results

Reporting the results of how these principles are used is another challenge. The architect's influence affects several key aspects of a project, including the ease of designing the solution, the efficiency with which the teams communicate and interact, the ability to deliver on time and within budget, and the satisfaction of stakeholders with the final solution. Measuring the productivity of a manufacturing process (such as number of widgets manufactured per day) is relatively straightforward and well understood. Measuring the productivity of creating software is inherently difficult. While many different standards are available, the one I emphasize here is source lines of code per staff month, or SLOC/SM. It is easily understood and provides a standard of comparison that can be used to measure progress against similar design efforts.

For the three design efforts noted earlier, the team's self-reported productivity was significantly greater than the industry average of 275 SLOC/SM for Web-based business projects [12]. The three teams reported the following:

- For the development of the product-ordering Web site, the team generated 2,370 SLOC/SM, deploying the site well ahead of schedule; previously dissatisfied stakeholders became strong advocates of the new process;
- The business-to-business integration team generated 3,760 SLOC/SM; the customer noted that of several companies involved in the integration, it viewed the architecture-driven process as "best in class"; and
- The design and development of the enterprise application proceeded smoothly, with the team generating 1,732 SLOC/SM; meanwhile, overall customer satisfaction with the development team improved dramatically.

Other factors (such as team size and the experience of individual team members) also influence productivity. However, the principles noted here strongly influenced each team to achieve much greater productivity than they expected. More important, customers realized significant improvement in terms of time to market and development costs.

Conclusion

Requirements and design teams, customers, and managers all look to the architect for leadership in the system-level design process. When the software-construction process begins, the architect must

proactively oversee the software's construction, particularly in large systems.

Designing a software solution involves the management of functional and nonfunctional requirements. Software architects must also be able to address the inherent complexity of building software as communicators and leaders and bring to bear proven skills related to the lower-level component design and construction tasks. Though some of these skills may be acquired through study, there is no substitute for hands-on experience.

These skills form a framework from which the architect, as well as other software professionals, may drive software projects toward success. In practice, it adds significant value to an organization, is vital to the growth of the software development profession, and represents the essence of the effective software architect. C

References

1. Alexander, C. *The Origins of Pattern Theory, the Future of the Theory, and the Generation of a Living World.* Speech at the 1996 ACM Conference on Object-Oriented Programs, Systems, Languages, and Applications (OOPSLA) (San Jose, CA, Oct. 6–10, 1996).
2. Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, Inc., 1978.
3. Brooks, F. *The Mythical Man-Month, Anniversary Edition.* Addison-Wesley, Boston, 1995.
4. Clavell, J., Ed. *The Art of War.* Delacorte Press, New York, 1983.
5. Fowler, M. *Patterns of Enterprise Application Architecture.* Addison-Wesley, Boston, 2002.
6. Fowler, M. *Is Design Dead?* Online article, 2001; www.martinfowler.com/articles/designDead.html.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
8. Glass, R. Sorting out software complexity. *Commun. ACM 45,* 11 (Nov. 2002), 19–20.
9. Harari, O. *The Leadership Secrets of Colin Powell.* McGraw-Hill, New York, 2002.
10. IEEE. *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std. 610.12-1990). IEEE, New York, 1990.
11. Object Management Group. *OMG Unified Modeling Language Specification.* OMG, Needham, MA, 2003.
12. Reifer, D. Industry software cost, quality, and productivity benchmarks. *Crosstalk: The Journal of Defense Software Engineering 7,* 2 (June 2004).

Matthew R. McBride (mcbride@computer.org) is a director of software development for Countrywide Financial Corp. and adjunct professor and advisory board member in the Department of Computer Science and Engineering at Southern Methodist University, Dallas, TX.