

Software Architecture

Maps encourage boldness. They're like cryptic love letters. They make anything seem possible.

Mark Jenkins, "To Timbuktu"

The software architecture of a system or a family of systems has one of the most significant impacts on the quality of an organization's enterprise architecture. While the design of software systems concentrates on satisfying the functional requirements for a system, the design of the software architecture for systems concentrates on the nonfunctional or quality requirements for systems. These quality requirements are concerns at the enterprise level. The better an organization specifies and characterizes the software architecture for its systems, the better it can characterize and manage its enterprise architecture. By explicitly defining the systems software architectures, an organization will be better able to reflect the priorities and trade-offs that are important to the organization in the software that it builds.

The software architecture of a system supports the most critical requirements for the system. For example, if a system must be accessible from a wireless device, or if the business rules for a system change on a daily basis, then these requirements drastically affect the software architecture for the system. It is necessary for an organization to characterize software architectures and the level of qualities that their systems support to fully understand the implications of these systems on the overall enterprise architecture.

Since this book has a focus on agile methodologies, it is important to discuss the relationship between software architecture and agile methodologies. The recent push in software development toward agile methodologies, such as eXtreme Programming (XP) (Beck 1999), in some ways counters the belief in an explicit and formal definition of software architecture. Many agile methodologists assert that software design is the result of iterative refactoring of a system by developers until a sufficiently workable design emerges. However, in XP these iterative refactorings are done in a small, easily understandable, conceptual framework for the system called the *system metaphor*. The system metaphor is a simple shared story for how the system works. It consists of the core concepts, classes, patterns, and external metaphors that

shape the system being built. The system metaphor plays the same role in the development of a system as conceptual software architecture. It provides a vision for the development of the software, and it is the goal that each system stakeholder must strive toward. A formal definition of the software architecture is more technical than the system metaphor, but both play the same role in providing a central concept for system development. The extent to which an organization needs to provide a formal technical description of the architectures of its systems depends on many factors. Clearly, it would not be cost-effective to formally define the software architecture for a small noncritical departmental software application, and it would be unacceptable to not formally define the software architecture of a highly available telecommunications software system. Each organization is different and has a different need for detailed software architecture. The agile principle of “If you don’t need it, then don’t do it” applies to software architecture as well as to all the other parts of an organization’s enterprise architecture. For more on agile methods and architecture, see Chapter 9, Agile Enterprise Architecture.

What Is Software Architecture?

No single standard definition of software architecture exists. However, many authors and researchers have attempted to define the term *software architecture*. Following are some of the most notable definitions:

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.” *Software Architecture in Practice*, Len Bass, Paul Clements, and Rick Kazman, Addison-Wesley, 1997.

“An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition.” *The UML Modeling Language User Guide*, Booch, Jacobsen, Rumbaugh, Addison-Wesley, 1999.

“Software architecture is a set of concepts and design decisions about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains.” *Software Architecture for Product Families: Principles and Practice*, Mehdi Jazayeri, Alexander Ran, Frank van der Linden, Addison-Wesley, 2000.

These definitions are a bit academic, and a practical guide must include a practical definition of software architecture and the reasons for having one. Mountains of research have been done on the subject, but why exactly would a project need to define the software architecture of the system it is building? With this practical guide in mind, we offer our definition of software architecture:

The software architecture of a system or a collection of systems consists of the important design decisions about the software structures and the interactions between those structures that comprise the systems. These

design decisions support a desired set of qualities that the system should support to be successful. The design decisions provide a conceptual basis for system development, support, and maintenance.

Creating software architecture is a difficult endeavor, and the software architect has one of the most difficult jobs in a software project. He or she must have the confidence of all the stakeholders. This confidence is based on a track record of successful projects and the respect of the developers who regard him or her as a technical leader. The architect must be able to communicate with varied constituencies. He or she must have excellent design skills, technology skills, and an understanding of software engineering best practices. He or she must be able to navigate through organizational politics to get the project done correctly and on time. The software architect must be a leader, a mentor, and a courageous decision maker.

Architecture, like life, is all about the people. Great people deliver great architecture, and the reverse is true. The techniques outlined in this chapter will give an architect some concepts and tips for delivering great architecture, but the best way to truly deliver great architecture is to start with truly great architects.

In cartography, a single map cannot fully characterize a place. There are many kinds of maps, such as highway maps, bike trail maps, and elevation maps. Each type explains and describes different aspects of the same physical place. Each map is relevant to a different user or stakeholder. The family on vacation is interested in having a highway map in its glove compartment. The cyclist needs the bike trail map, and a mountain climber needs the elevation map. In addition, a map doesn't have to be perfectly accurate to be useful. If a perfect scale map of rides and paths were given to visitors to Walt Disney World, it would be accurate and somewhat useful. However, the map that is actually handed out at Walt Disney World contains pretty pictures of the interesting rides and the paths are not to scale. For the Disney World visitor, this map is more interesting, though useful, and it imparts a better understanding of the layout of the park than a cartographer's map would. The same holds true in software architecture. The architecture that is presented to the end-user contains less information and is more interesting, but focuses on the critical aspects of the system that the particular stakeholder community should understand. The software architecture that is needed by developers can be more like a cartographer's depiction of the software architecture, with multiple maps outlining the multiple aspects of the system in more detail.

Just like maps, the purpose of software architecture is to impart an understanding of the design of the system to the reader. The point of software architecture is to communicate an idea. It takes the reader into the software and explains the important concepts. It helps them understand the important aspects of the system and gives them a feeling for a system without actually having to see inside it.

Despite the invention of satellite imagery, maps are still important. We can now get an exact picture of Earth at any level of detail we need. The pictures show rivers, oceans, and more, as well as the layout of Walt Disney World in their exacting detail.

The Role of a Software Architect

Why We Need Software Architecture

“Maps have the character of being textual in that they have words associated with them, that they employ a system of symbols within their own syntax, that they function as a form of writing (inscription), and that they are discursively embedded within broader contexts of social action and power.” John Pickles, “Text, Hermeneutics and Propaganda Maps,” in Trevor J. Barnes and James S. Duncan, eds., *Writing Worlds: Discourse and Metaphor in the Representation of Landscape*, London: Routledge, 1992, 193.

Maps and architecture don't describe reality. They are representations of reality within a chronological and cultural context. They also have a distinct perspective on the place they describe. In other words, reverse engineering a Unified Modeling Language (UML) diagram from the source code of the system does not create an architectural model. The architectural model communicates the important design decisions that were made at a particular time and that were important to a particular group of people.

The cartography metaphor works, except for the fact that architecture is an upfront activity. In cartography, maps are created for existing places. They are meant to describe something that has already been created. In software architecture, the maps or models depict software that is yet to be created. The models embody the important design decisions that have been made about the system. By creating and contrasting different models of software that fulfill the same purpose, intelligent decisions can be made about which designs are better than others. This is the second purpose of software architecture.

The Two Extremes

Software development approaches vary between two extremes. The first method involves little or no upfront modeling or design. This is the “shanty town” method of system development in which a few developers code without a mental picture in their heads about the system they are building. Also, some managers believe that if developers aren't coding, they aren't working. These project managers also believe that the sooner developers begin coding, the sooner they will be done. This stems from the incorrect belief that a constant amount of time is involved in the coding of the system no matter what upfront process is used. In this type of environment, developers don't fully understand the requirements for the system. Some of these environments deliver decent software through heroics by developers and frequent rewrites, although this approach is not repeatable, and it is extremely risky.

The other extreme in software development is “ivory tower” software architecture in which a design team or a single architect design a system in every detail, down to the class and method level. The architect has a clear picture in his or her head about the design of the system but has not left many of the implementation details to the developers. The belief in these environments is that the architects are the most experienced developers and thus can design the best possible system from start to finish. No one person or small team can possibly understand all the requirements, predict every change in requirements, and have expertise in every technology that the project is built upon. Developers in these environments also suffer from low morale because they are perceived as somehow inferior to the designers of the system. Morale is also poor because the developers must implement the design in a prescriptive way with little or no input into the design of the system.

The Middle of the Road

So what does all this have to do with software architecture? Software architecture is the middle road between no design and complete design. It is a view of the system design that shows how the design satisfies the critical requirements of the system. It is the role of the software architect to design the structures of the software such that those critical requirements are satisfied. It is also the goal of the software architecture to facilitate the development of the system by multiple teams in parallel. In addition, if multiple teams or departments within an organization will support and maintain the software, the software architecture will allow those parts of the system to be managed and maintained separately. The most important role that the software architecture has is that of an organizing concept for the system. The software architect has an idea how the system should work. The software architecture is the communication of that idea to other system stakeholders so that everyone understands what the system does and how it does it.

In a practical sense, two rules determine whether or not a design detail should be included in the software architecture:

1. The design detail must support a quality requirement.
2. The design detail must not detract from stakeholder understanding of the software architecture.

If the design detail does not somehow improve on a quality requirement of the system, it should be left out. For example, the architect might not choose XML due to performance concerns. However, the system might benefit more from the modifiability of XML. Without an elicitation of the quality requirements for the system, these types of decisions might be made based on personal preference instead of quality requirements. Also, if a design detail is complex and cannot be described in a simple way that stakeholders can understand, it should not be included in the architecture. A design detail that is not understandable is also a sign of a bad design.

Many people believe that the software architecture is meant only for developers to use as an overall guide for system design and construction. While this may be the software architecture's primary purpose, other system stakeholders can use the architecture as a basis to guide their activities as well. The following are some of the system stakeholders:

- Developers
- Managers
- Software architects
- Data administrators
- System customers
- Operations
- Marketing
- Finance
- End-users

The System Stakeholders

- General management
- Subcontractors
- Testing and quality assurance
- UI designers
- Infrastructure administrators
- Process administrators
- Documentation specialists
- Enterprise architects
- Data administrators

The software architect must elicit input from all the system stakeholders to fully understand the requirements for the architecture. This is important because the requirements are built from the perspective of what the system should do. However, the architecture must reflect how the system will perform those functions.

The system's customers want the system to be of high quality. They want the system to be delivered in a timely manner. And they want it to be developed as inexpensively as possible.

The development organization is looking for a vision for the system it is going to design and develop. It wants to know that the architecture is easy to implement. It has hard deadlines that it must meet, so reusability is important. The developers are going to be looking for technologies in the architecture that they currently understand. They want the architecture to match their desired platforms, development tools, libraries, and frameworks. They need to meet dates, so the architecture should ease their development effort. Most of all, they want an architecture that they have participated in developing and evolving throughout the lifetime of the product.

The operations group wants a product that is supportable and maintainable. The product must not fail. It has to meet service level agreements that it can only meet if the product is reliable. If the system goes down, this group is on the front lines trying to get it back up. When it does fail, the operators need to find out why so that the problems can be fixed. Therefore, the system should provide some traceability for system transactions so that what went wrong can be tracked down. The operations group also has the responsibility for installing new machines and upgrading software platforms on a schedule. It needs the software to be flexible and portable so that moving it to a new machine or onto a new version of the operating system is fast and easy.

The marketing people are looking for an architecture that can deliver the greatest number of features in the shortest amount of time. An architecture that is flexible and can integrate off-the-shelf packages will win their hearts. In addition, if the product is a commercial product, the architecture may be a key selling point. Savvy customers of commercial products will recognize a good architecture from a bad one.

End-users need great performance from the system. It must help them get their jobs done more quickly and easily. The system must be usable. The interfaces must be designed with end-user tasks in mind and, ideally, the

system should be customizable so that end-users can choose how they wish to use it.

Every stakeholder has his or her perspective on what is important for the system to do. It is up to the architect to mediate among these individual concerns. Not all stakeholders can get everything they want all the time. Sometimes, a requirement can be easily met without detracting from another requirement. Sometimes trade-offs have to be made among the various requirements. These are the decisions that the architect must facilitate within the constraints that the project sponsor places on time and money.

Our fictitious automobile manufacturer Canaxia was losing sales to other manufacturers because it relied completely on its dealer network to sell its cars. Other auto manufacturers created Web sites that allow customers to customize and order their car online. The other auto manufacturers ship the cars from their manufacturing facilities directly to the customers through local dealerships. This resulted in greater customer service and reduced the hassle of buying a car from a dealership and struggling with the sales process.

Canaxia was about two years behind their competition. The management of the company has decided to fund a project for twelve months to create a Web site and related infrastructure that exceed the capability of their competition.

Canaxia has a mainframe system for orders, inventory, and financials. Its inventory and order system was written in the 1980s. The system functions, but it is very expensive to change. The financial system was purchased and customized in the late 1990s. The infrastructure architects have decided that IBM MQSeries will be used to integrate all the enterprise systems. The standard for Web development in the company is Java and J2EE.

The architects on the project faced a difficult job. They realized that the software architecture is implemented at the beginning, middle, and end of every project. However, much more emphasis is on it at the beginning of every project. Before the architects started, they created a checklist of principles they would strive to follow while they created the architecture:

1. The architecture should be thin.
2. The architecture should be approachable.
3. The architecture should be readable.
4. The architecture should be understandable.
5. The architecture should be credible.
6. The architecture doesn't have to be perfect.
7. Don't do big upfront design. If given a choice between making the model perfect or implementing it, implement it.
8. Do the simplest thing that could possibly work without precluding future requirements.
9. The architecture is a shared asset.
10. Involve all stakeholders but maintain control.
11. The architecture team should be small.
12. Remember the difference between a pig and a chicken.

Creating a Software Architecture: An Example

The Pig and the Chicken

One day on a farm near Canaxia, a pig and a chicken decided to open a restaurant.

The pig turned to the chicken and asked him, "So what should we call this restaurant?"

The chicken replied, "How about Ham 'n' Eggs?"

The pig thought for a moment and said, "I don't think that's a very good idea. You would be involved, but I would be committed."

The Scrum (Schwaber et al. 2001) development methodology uses this story as the central theme for distinguishing between those people who are pigs (assigned work) and those who are chickens (interested, but not working).

When creating software architecture, chickens can wreck the process in two ways. If chickens are on the architecture team, you should just give up and open a restaurant. Assuming that the architecture team consists of all pigs, make sure that the architecture addresses the concerns of the pigs in the organization. Don't let the chickens sneak in an egg or two and make you design for something that isn't really important.

The following sections describe the steps that the architects at Canaxia performed to create their architecture (Bass, Clements, and Kazman 1997).

The Business Case

Canaxia was lucky; it involved the architecture team when it created the business case for the system. Many organizations do not involve an architect at project conception. The architecture team was involved from the start of the project. The business customers needed to understand the relative costs of the various solutions they were contemplating. The architecture team understood the current technical environment and the tools, processes, and personnel required to implement the various options presented by the business community. The business customers realized that a great solution that is late is sometimes worse than a mediocre solution that is delivered quickly. Only the architecture team can evaluate the various options and provide input into the time required to create each one.

At Canaxia, the architect was involved up front with auto dealers, management at the company, system end-users, and the development team. The business case took into consideration the timeframe and the technical complexity involved in creating a Web site for the automobile business.

Understanding the Requirements

Canaxia realized that an architect couldn't build architecture for a system that he or she didn't understand. The requirements of the stakeholders provide the architect with a context from which to create a design. A use case is a common way of capturing a requirement. The use case describes some action that the system must perform. A use case describes in detail every input/output interaction that the system performs with a user or another system. These are

known as *functional requirements*. In addition, the use cases specify how the interaction will occur. These are modifying clauses known as “nonfunctional requirements.” For example, the *use case*, “User enters amount, presses enter, and system displays the invoice” is a functional requirement. A nonfunctional requirement modifies this phrase, such as “User enters amount, presses enter, and system displays the invoice within 5 seconds.” The “within 5 seconds” is a nonfunctional or quality requirement of the system.

The architect must document how the system will accept the amount described in the use case by using a user interface, validating the request, storing the data in a database, and generating an invoice for the user. There is no architecture for the use case, but the architecture must satisfy all the use cases, including the nonfunctional or quality requirements. The architecture should address each pattern of interaction, such as “User enters data, data are saved to database, a result or an error is displayed.”

At Canaxia, the requirements were well documented, but even more importantly, user representatives were involved in every stage of the project and provided daily input into the architecture and development of the system.

Creating or Selecting the Architecture

The requirements gathering process delivers a set of functional requirements with qualities identified with each requirement. At Canaxia, these requirements were a set of use cases or user stories. Each use case had a set of qualities that needed to be supported by the use case. The architecture that Canaxia selected needed to support all these requirements and qualities. Some of the requirements were easily met using techniques, technologies, and practices with which the developers and architects were familiar. The difficult part of selecting the architecture was satisfying those qualities using techniques and technologies that were risky and unknown.

To address and reduce the technical risks in the project, the architects created an *architecture baseline*. This was the main milestone in the development of the architecture (Jacobsson, Booch, Rumbaugh 1998). The architecture baseline is the first fully executable portion of the system. A small thread of execution through all system layers was completed to prove that the system could be built. To create an architecture baseline, the architecture team followed the steps described in the following paragraphs. These steps were performed in an iterative manner with the architects working toward an architecture baseline that satisfied the functional and quality requirements for the system:

Select Use Cases

Select a small number of use cases from the requirements for the system. These are selected to address the most technically risky areas of the system. This subset of use cases is the *architecturally significant* use cases. For example, Canaxia chose a new software package for printing documents from the system. It included the use cases for printing documents in the architecture baseline to demonstrate that the interface to the new software package was going to work.

Identify Important Qualities

Software architecture requires trade-offs. Canaxia could not get the architecture to satisfy every requirement in the best possible way. Therefore, the qualities that the architecture should support were prioritized. For example, Canaxia valued performance over modifiability. When decisions required a trade-off between quality attributes, it was helpful to understand what quality attributes were desired more than others in the system.

Design the Architecture

Canaxia designed an architecture that was able to implement the use cases that were chosen for the system. This involved adopting one or more *architectural styles* (described later in this chapter) and fleshing out those styles into a more detailed design. Canaxia decided to use the most well-known architectural style, the model-view-controller architectural style. Canaxia knew it was done with this step when the decisions it was making became harder and harder because less tangible information on which to rely was available.

Set Up a Development Environment

Canaxia then set up a development environment that included:

- Servers and space on servers such as file servers, application servers, and database servers
- Modeling and drawing tools
- Whiteboards
- Digital camera for taking pictures of whiteboards
- Project Web site and file subfolder
- Software with which they needed to integrate
- Integrated development environment (IDE)
- Unit testing framework (XUnit)
- Version control software
- Automated build process

Implement the Design

Canaxia started implementing the design. This was done by first implementing the tests for the design. Writing the tests fleshed out the contracts in the modules. When Canaxia began implementing the design, it learned what did and didn't work. Implementation will prove what was just theory in the design and will provide a concrete understanding of the architecture being developed. Canaxia stopped implementing when it exhausted the initial designs. When this happened, Canaxia improved the design and implemented more of the architecture in several more iterations until they were happy with it.

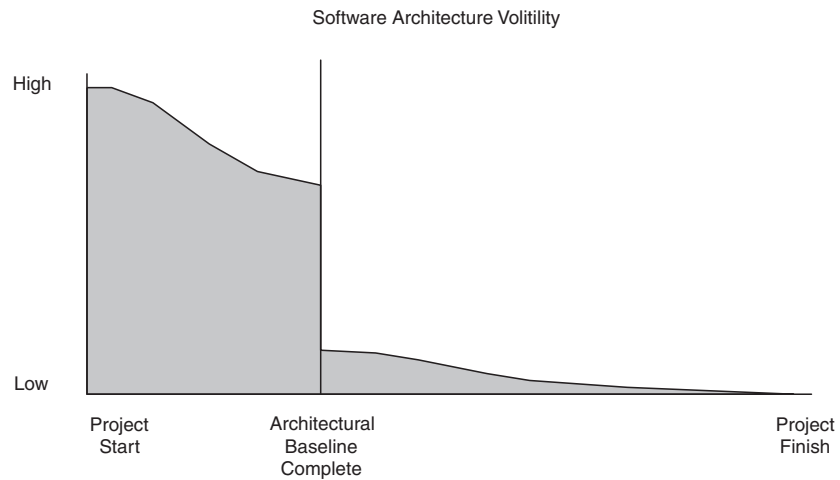
Know When You Are Happy with the Design

The architecture team tried to establish how to know whether or not the architecture was a good one. It adopted the following checklist of early warning signs when things might be going astray (Abowd et al. 1996):

1. The architecture is forced to match the current organization. Sometimes a good architecture is changed because someone says, "We just don't do that here." Architecture can be polluted because the practices and technical infrastructure of the organization don't mesh with the architecture for the project. Many times, compromises in the architecture are necessary to ensure enterprise consistency, but the conceptual integrity of the architecture should be maintained as much as possible.
2. There are too many top-level architectural components (too much complexity). If the number of components reaches a level (usually more than 25), the architecture becomes too complex for it to have conceptual integrity. The roles and responsibilities of each component must be clear. When the system consists of too many components, the responsibilities of the component tend to blend into each other.
3. One requirement drives the rest of the design. Some projects are built with a single overriding goal in mind that is sometimes the pet requirement of the project sponsor, project manager, or project architect. When the system is built around a single overriding requirement, other requirements may not be addressed.
4. The architecture depends on the alternatives presented by the operating platform. Sometimes the platform chosen is the wrong one for the project. For example, a project that requires a high level of usability might not be possible if the platform chosen for the user display is a 3270 terminal.
5. Proprietary components are used instead of equally good standard ones. Architects and developers are often lured by the latest cool technology or interesting designs that may have alternative standard technology available. For example, projects should take advantage of the capabilities of application server technology. When developers begin creating load balancing schemes or persistence frameworks, it is usually an indication that there is a problem. It would be better to use the standard capabilities of an application server and spend time developing software to solve the business problem rather than creating new technical features.
6. The component division definition comes from the hardware division. The components should be designed without regard for the physical topology of the system, except to recognize that a network is involved in transmitting data between components. The components should map to a discrete business or technical function of the application that is being built. These components should take advantage of the scalability features of modern computing systems.

Figure 2-1

Software architecture volatility throughout a software development project.



7. The design is exception driven; the emphasis is on the extensibility of the system rather than on the core requirements for the system. By “exception driven,” we do not mean exception handling but rather that the system should not be designed exclusively with the requirements that someday might be necessary in mind. It should be designed to solve the more immediate requirements.

The architecture team finished the baseline architecture and felt good about it. It was careful to notice “code smells” during the creation of the baseline. These were aspects of the code that just didn’t feel right. When the team saw something sneak in that didn’t really belong or required extensive refactoring, it had the courage to rip it out and start over.

As shown in Figure 2-1, the architecture is volatile until the architecture baseline is complete. During the period of time that the architecture baseline is worked out, major design decisions will change. During this period, the overall design is proven. When the architectural baseline is complete, the architecture for the system should settle down so that full development can be done based on the stable baseline. If the architecture is volatile throughout the development process, the project will likely fail.

Representing and Communicating the Architecture

The architects at Canaxia knew for the architecture to be effective, it needed to be effectively communicated to all the stakeholders of the system. After all, architecture really has no other purpose than to provide a concept for the system about to be built. The architecture needed to be clear, concise, and understandable so that the important concepts within the architecture could be implemented and supported by the stakeholders. The architects understood that it was these important concepts that in many ways determined the success or failure of the project at every stage of its lifecycle.

To effectively communicate the architecture to the various stakeholders, the team created targeted material for each stakeholder community. For example, the user community was very interested in the usability of the sys-

tem and how the architecture would support their needs. Therefore, the architects provided a storyboard and a prototype to describe that. This was a very effective means of conveying those aspects of the architecture to this community. The support organization wanted to know how to maintain the system, so a discussion of the software design using UML and data models was an appropriate means of conveying this information to them.

Analyzing and Evaluating the Architecture

Throughout the development of the system, Canaxia was careful to continuously evaluate the architecture to make sure that it met the needs of the project. It knew that the architecture really isn't finished until the project is delivered, and perhaps not even until the system is retired. It used an iterative process of evaluating and evolving the architecture to improve it as necessary throughout the system life cycle. The architecture evaluation process in the initial phases of a project was done through scenario-based techniques. To evaluate software architecture, Canaxia identified the architecturally significant scenarios for the architecture to support. Some methodologies, such as XP, call these scenarios *user stories*. These scenarios provided a context from which the quality attributes of the architecture could be estimated. As the project developed, the estimates became measurements and test cases for verifying that the desired quality attributes were actually supported by the software that was being created. For example, when the architecture was created, the team estimated the level of performance that the architecture should support based on the number of computations and network hops. Once the architecture baseline was completed, the estimates became measurements of performance for the set of scenarios that were implemented. If the performance requirement was not met or exceeded, the architecture was redesigned to support the requirement.

At each stage of the development process, the architecture was evaluated. At later stages in the project, some estimates were very precise because they were based on real working code. However, some of the quality attributes were difficult to measure and were somewhat subjective. For example, it was easy to use a stopwatch to measure the performance of a transaction through the system, however estimating the maintainability of the system was much more difficult. The maintainability estimation was based on many criteria that included the technology that was used, whether or not the system was sufficiently modular, and how much configuration data were used to configure the system at run-time, as well as many other criteria particular to the system that was built. Estimating this even after the system was built was difficult and still somewhat subjective. Canaxia had to rely on the architecture team's skill and experience within the organization and with software designs to deliver the right level of maintainability for the project.

Canaxia considered three popular methodologies for evaluating software architectures (Clements, Kazman, and Klein 2002):

1. Architecture Trade-off Analysis Method (ATAM)
2. Software Architecture Analysis Method (SAAM)
3. Active Reviews for Intermediate Designs (ARID)

Canaxia understood that for critical projects that require a large amount of rigor during the development process, formal methods help to formally evaluate the software architecture of a system. The ATAM and SAAM methods for evaluating software architectures are comprehensive, while the ARID method is meant for intermediate design reviews to ensure that the architecture is on track throughout the project. Canaxia purchased the book *Evaluating Software Architecture* (Clements, Kazman, and Klein 2002) to learn about each method in depth. In the end, Canaxia created its own ad hoc method to evaluate the software architecture that was partially based on the three methods presented in the book.

Ensuring Conformance

The architecture team realized that even with the best architecture, development teams don't always implement an architecture correctly. Usually, this is the fault of a poor architecture or poor communication of the architecture to developers. In order for the architecture to matter, the architecture in concept must become executing code in a system. At Canaxia, the architecture was implemented correctly in the final system. All the desired system qualities were met and the system was delivered on time and under budget.

In addition, the architects and developers realized that the architecture is an ongoing effort in refinement, even after the project is completed. The architects understood they needed to be involved not only during design and construction but also during maintenance of the system.

At Canaxia and at every other company that is developing software, the question of how software architecture should be represented is a difficult one. Stakeholders of the system need targeted material that speaks to them because, above all else, stakeholders must understand the architecture in order to implement it correctly. For technical stakeholders or those who understand software development, UML is the most popular notation to describe the design of software systems. On the surface, UML appears to be well suited as a notation for describing software architectures. UML has a large set of elements that can be used to describe software designs. The Rational Unified Process (RUP) best represents the viewpoint that UML is adequate for representing software architectures.

RUP is an "architecture-centric" process that promotes the use of UML for depicting software architecture. However, academics have questioned UML as a means of depicting software architecture (Medvidovic 2002). This is mostly because early versions of UML did not contain notations for components and connectors as first-class elements. Other constructs are important when describing software architecture, such as ports or points of interaction with a component and roles or points of interaction with a connector. In UML 1.5, components are expected to be concrete, executable software that consumes a machine's resources and memory. Although some parts of software architecture are concrete components, not all architectural components are concrete software entities. They may be an entire system or a federation of systems that need to be depicted as a single component that provides a single function.

Architecture Description Languages and UML

UML 2.0 moves components and connectors through the lifecycle and addresses many of these concerns.

In addition, a connector is an architectural notion that describes something that provides a conduit from one or many components to one or many components. The connector may also adapt the protocol and format of the message from one component to another. UML 1.5 does not provide a similar concept, although one could get around this by depicting a connector as a set of cooperating classes or as a component.

UML 2.0 was released in June 2003 and addresses many of these issues in the language. In UML 2.0, a component can be depicted to use an interface and provide a port. A port can be a complex port that provides and consumes multiple interfaces. Each interface can be designated with attribution that indicates the visibility of the interface, whether or not it is a service and has asynchronous capability or not. UML 2.0 is making strides toward becoming the standard notation for depicting architectures.

Using UML has the distinct advantage of being a standard notation for software design. It is desirable to use UML for describing software architecture because it is standard. To use earlier versions of UML for this purpose, readers of the UML architecture description must be willing to suspend their disbelief. For example, an architectural component could be described by a UML component in a UML diagram. Another popular approach is to use a stereotyped class. This will work as long as the reader does not take this to mean that the component is an actual software entity and that it describes a conceptual component, not a physical one.

In addition, researchers have created several architecture description languages that have a small set of core elements that allow for the first-class representation of architectural concerns. These languages focus on a precise description of software architecture. They argue that the only way to assess the completeness and correctness of the software architecture is to precisely describe the software architecture. These methods and languages suffer from the fact that there are dozens of them, and they all have the goal of precision over understandability.

The bottom line is that there is no standard notation for documenting software architecture. The key criteria when choosing a modeling language—whether it is UML, an ADL, or your own notation—is that it should further the understanding of the architecture by those who read it. The model should accomplish this primary goal regardless of the notation used to document it.

The software architecture of a system promotes, enforces, and predicts the quality attributes that the system will support. Quality attributes are those system properties over and above the functionality of the system that make the system a good one or a bad one from a technical perspective. There are two types of quality attributes: those that are measured at run-time and those that can only be estimated through inspection. Since the software architecture of a system is a partial design of a system before it is built, it is the responsibility of the software architect to identify those quality attributes that are most important and then attempt to design an architecture that

Quality Attributes

Software Architecture
49

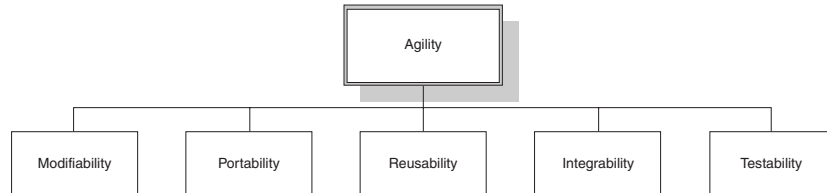
reflects those attributes. The quality attributes that most architects should be concerned with are (Bass, Clements, Kazman 1997; Clements, Kazman, Klein 2002):

1. **Performance**—a measurement of the system response time for a functional requirement.
2. **Availability**—the amount of time that the system is up and running. It is measured by the length of time between failures, as well as by how quickly the system is able to restart operations after a failure. For example, if the system was down for one day out of the last twenty, the availability of the system for the twenty days is $19/19+1$ or 95 percent availability. This quality attribute is closely related to reliability. The more reliable a system is, the more available the system will be.
3. **Reliability**—the ability of the system to operate over time. Reliability is measured by the mean-time-to-failure of the system.
4. **Functionality**—the ability of the system to perform the task it was created to do.
5. **Usability**—how easy it is for the user to understand and operate the system.
6. **Security**—the ability of the system to resist unauthorized attempts to access the system and denial-of-service attacks while still providing services to authorized users.
7. **Modifiability**—the measurement of how easy it is to change the system to incorporate new requirements. The two aspects of modifiability are cost and time. If a system uses an obscure technology that requires high-priced consultants, even though it may be quick to change, its modifiability can still be low.
8. **Portability**—measures the ease with which the system can be moved to different platforms. The platform may consist of hardware, operating system, application server software, or database server software.
9. **Reusability**—the ability to reuse portions of the system in other applications. Reusability comes in many forms. The run-time platform, source code, libraries, components, operations, and processes are all candidates for reuse in other applications.
10. **Integrability**—the ability of the system to integrate with other systems. The integrability of a system depends on the extent to which the system uses open integration standards and how well the API is designed such that other systems can use the components of the system being built.
11. **Testability**—how easily the system can be tested using human effort, automated testing tools, inspections, and other means of testing system quality. Good testability is related to the modularity of the system. If the system is composed of components with well-defined interfaces, its testability should be good.

12. **Variability**—how well the architecture can handle new requirements. Variability comes in several forms. New requirements may be planned or unplanned. At development time, the system source code might be easy to extend to perform new functions. At run-time, the system might allow pluggable components that modify system behavior on the fly. This quality attribute is closely related to modifiability.
13. **Subsetability**—the ability of the system to support a subset of the features required by the system. For incremental development, it is important that a system can execute some functionality to demonstrate small iterations during product development. It is the property of the system that allows it to build and execute a small set of features and to add features over time until the entire system is built. This is an important property if the time or resources on the project are cut. If the subsetability of the architecture is high, a subset of features may still make it into production.
14. **Conceptual integrity**—the ability of the architecture to communicate a clear, concise vision for the system. Fred Brooks writes, “I am more convinced than ever. Conceptual integrity is central to product quality. Having a system architect is the most important single step toward conceptual integrity. . . . After teaching a software engineering laboratory more than 20 times, I came to insist that student teams as small as four people choose a manager and a separate architect” (Brooks 1995). Kent Beck believes that metaphors are the most important part of the eXtreme Programming methodology (Beck 1999). The metaphor is a powerful means of providing one or more central concepts for a system. The metaphor provides a common vision and a shared vocabulary for all system stakeholders. The metaphor provides a means to enforce conceptual integrity. When the design of the system goes outside the bounds of the metaphor, the metaphor must change or new metaphors must be added; otherwise, the design is going in the wrong direction. If any of these design decisions are made without the concept feeling right, the conceptual integrity of the system will be lost. Sometimes the system metaphor is an architectural pattern, such as MVC or Blackboard (discussed later in this chapter). These architectural patterns provide a common metaphor for system developers or others who understand the patterns. However, they don’t help the stakeholders who aren’t familiar with the patterns. One good thing about using architectural patterns for the system is that they describe the structures of the software in more detail; on the downside, not all stakeholders will understand the references.
15. **Buildability**—whether or not the architecture can reasonably be built using the budget, staff, and time available for delivery of the project. Buildability is an often-overlooked quality attribute. Sometimes, the best architects are simply too ambitious for a project team to complete given project constraints.

Is Agility a Quality Attribute?

Some architects use the term agile to describe their architecture. While flexibility and agility are important, these words have many dimensions. If the architecture is agile, does that mean that it is easily changed? Is it easy to integrate it into the enterprise? Is it portable, reusable, testable? The answer to all these questions is probably "yes." Agility is a composite quality that includes many of the base quality attributes. If the levels of maintainability, portability, testability, and integrability are high, the architecture is most likely an agile one.



It is important for the system and enterprise architecture designers to understand the desired qualities that the systems must exhibit. To do so, the architects should encourage as precise a specification of the desired quality attributes as possible. A checklist provides a way of ensuring the completeness of the specification. Following are some of the questions to ask in order to fully characterize the desired quality attributes for a system (Clements, Kazman, and Klein 2002; McGovern, Tyagi, Stevens, and Mathew 2003):

Performance

- What is the expected response time for each use case?
- What is the average/max/min expected response time?
- What resources are being used (CPU, LAN, etc.)?
- What is the resource consumption?
- What is the resource arbitration policy?
- What is the expected number of concurrent sessions?
- Are there any particularly long computations that occur when a certain state is true?
- Are server processes single or multithreaded?
- Is there sufficient network bandwidth to all nodes on the network?
- Are there multiple threads or processes accessing a shared resource? How is access managed?
- Will bad performance dramatically affect usability?
- Is the response time synchronous or asynchronous?
- What is the expected batch cycle time?

How much can performance vary based on the time of day, week, month, or system load?

What is the expected growth of system load?

Availability

What is the impact of a failure?

How are hardware and software failures identified?

How quickly must the system be operational after a system failure?

Are there redundant systems that can take over in case of a failure?

How do you know that all critical functions have been replicated?

Are backups done? How long does it take to back up and restore the system?

What are the expected hours of operation?

What is the expected up-time per month?

How available is the current system? Is this acceptable?

Reliability

What is the impact of a software or hardware failure?

Will bad performance impact reliability?

What is the impact of an unreliable system on the business?

Can the integrity of the data be compromised?

Functionality

Does the system meet all the functional requirements specified by the users?

How well can the system respond and adapt to unanticipated requirements?

Usability

Is the user interface understandable?

Is the interface adaptable to support the needs of people with disabilities?

Do the developers find the tools provided for creating the system usable and understandable?

Portability

Do the benefits of a proprietary platform outweigh the drawbacks?

Can the expense of creating a separation layer be justified?

At what level should system portability be provided? At the application, application server, operating system, or hardware level?

Reusability

Is this system the start of a new product line?

Will other systems be built that more or less match the characteristics of the system under construction? If so, what components will be reused in those systems?

What existing components are available for reuse?

Are there existing frameworks or other code assets that can be reused?

Will other applications reuse the technical infrastructure that is created for this application?

Is there existing technical infrastructure that this application can use?

What are the associated costs, risks, and benefits of building reusable components?

Integrability

Are the technologies used to communicate with other systems based on standards?

Are the component interfaces consistent and understandable?

Is there a process in place to version component interfaces?

Testability

Are there tools, processes, and techniques in place to test language classes, components, and services?

Are there hooks in the frameworks to perform unit tests?

Can automated testing tools be used to test the system?

Can the system run in a debugger?

Subsetability

Is the system modular?

Are there many dependencies between modules?

Does a change in one module require a change in other modules?

Conceptual Integrity

Do people understand the architecture? Are there too many basic questions being asked?

Is there a central metaphor for the system? If so, how many?

Was an architectural style used? How many?

Were contradictory decisions made about the architecture?

Do new requirements fit into the architecture easily, or do new features require “code smells”? If the software starts to “stink,” the conceptual integrity has probably been lost.

Buildability

Are enough time, money, and resources available to build an architecture baseline and the project?

Is the architecture too complex?

Is the architecture sufficiently modular to promote parallel development?

Are there too many technical risks?

This is by no means an exhaustive list of questions to ask about architecture or a design. On every project, there are specific questions to ask about the domain of that project and organization. For example, if an organization uses messaging middleware, there is a list of very specific questions about how that middleware system is used and whether or not the architecture uses it effectively and correctly. If the organization has an in-house framework for creating components and services, there is a list of questions about the design of a component or service that uses that framework. The questions that must be asked on each project vary. It is important that the architecture team understand the intricacies of the organization’s domain, the architecture that supports it, and, especially, the obstacles that may be encountered so that they can be avoided. Only if a design team is aware of the details of a particular organization can it properly design a system that runs in that organization.

Nonfunctional Requirements and Quality Attributes

Nonfunctional requirements are almost the same as quality attributes. Nonfunctional requirements are defined as the nonobservable properties of the system. A classical functional requirement can be stated as a noun–verb phrase, such as “The system displays the report.” A nonfunctional requirement can be thought of as a modifying clause, such as “The system displays the report within 5 seconds.”

In other words, a functional requirement is stated in mathematical terms. An input is given to a function, and the function returns an output. The nonfunctional requirements state how the functions of the system will be performed.

The big problem with nonfunctional requirements is that most requirements gathering teams tend to leave these requirements out, or they do not fully state the nonfunctional requirements of the system. They do this because they focus only on the business case for the system. They concentrate on the business stakeholders and not on the technical stakeholders of the system. Requirements such as modifiability, security, and so on are usually overlooked because they are not perceived as directly impacting the cost-benefit for the system. These requirements are perceived as being only technically related requirements. This is not true. Nonfunctional requirements are critically important to the success of any project. They are technical risks that, if not addressed, can make or break a system. For example, if a report displays

correctly but it does so in thirty minutes instead of a more reasonable thirty seconds, the functional requirement is met but the system is not usable because it does not meet the user's performance expectation. If a hacker is able to retrieve credit card information for the businesses customers, for example, the future of the company may be in jeopardy.

Nonfunctional requirements are not only within the jurisdiction of the project under development. The quality attributes of a system are supported at the enterprise level. Typically, the reliability, availability, performance, and other quality attributes are supported by the platform on which the project is deployed.

Architectural Viewpoints

Once the quality attributes of the system have been characterized, how do you document how they will be fulfilled? Creating the software architecture of a system can be a large endeavor, depending on the size and complexity of the system. The software architecture should be partitioned into multiple views so that it is more understandable. When we describe any entity—be it an architecture, a process, or an object—there are many different perspectives or viewpoints to describe. For example, if you were to describe an orange to someone, would you talk about its color, weight, sweetness, skin thickness, or some other attribute of the orange? The attributes that you would choose to describe would depend on the role, the needs and the perspective of the person to whom you are describing the orange. If the person is a purchaser at a grocery store, the purchaser would be more interested in the color of the orange so that customers would buy the orange. A consumer of oranges would be more interested in the taste of the orange than the color. A chemist that is considering using oranges in a cleaning solution product would be interested in the chemical properties of oranges that could be applied to cleaning solutions. The point is that every stakeholder in oranges has a different need and a different perspective, and thus the orange must be described to them in different ways.

The same goes for architecture. All stakeholders in the architecture hold viewpoints that speak to their different perspectives on the project and within the organization. It is also important to separate the architecture description into different viewpoints so that a single stakeholder can understand different aspects of the architecture.

4+1 View Model of Software Architecture

No single standard set of viewpoints must be described for software architecture. In fact, we discuss two popular models to illustrate the point that multiple views are necessary. Philippe Krutchen from Rational software divides architecture into the 4+1 view model (Krutchen 1995) discussed in the following sections.

Logical View

The logical view, or logical architecture, is the object model for the design. It describes the structures of the software that solve the functional requirements for the system. It is a subset of all the classes of the system. The logical view is strictly a structural view of the software, including the important classes and class relationships in the architecture.

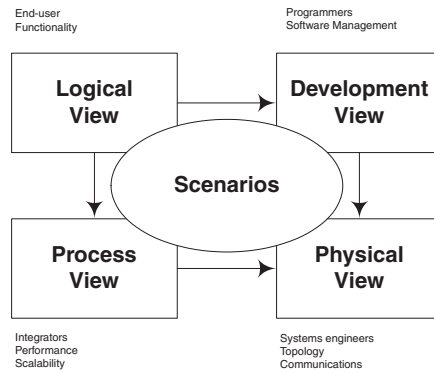


Figure 2-2
4+1 view.

Krutchen, P. *The 4 + 1 View Model of Architecture*. IEEE Software, pp. 42–50, November 1995.

Process View

The process view, or process architecture, describes the view of the architecture that includes running processes and instantiated objects that exist in the system. It describes important concurrency and synchronization issues.

Development View

The development architecture view focuses on the module organization of the software. It shows how classes are organized into packages, and it outlines the dependencies between packages of software. This is the view of the design that shows the layered structures of the software and what the responsibilities of each layer in the system are.

Physical View

The physical view describes the machines that run the software and how components, objects, and processes are deployed onto those machines and their configurations at run-time, see Figure 2-2.

+ 1

The +1 in the 4+1 view model describes the scenarios that the architecture is meant to satisfy. The scenarios represent the important requirements that the system must satisfy. The scenarios that are chosen are those that are the most important to solve because they are either the most frequently executed or they pose some technical risk or unknown that must be proven out by the architecture baseline. The other four views are centered on the set of scenarios that are chosen for the creation of the architecture.

Applied Software Architecture Viewpoints

The 4+1 view model is only one of several suggestions for the viewpoints that should be included in software architecture. Hofmeister, Nord, and Soni (1999) propose a different set of viewpoints.

Conceptual Architecture View

The conceptual architecture view is similar to the logical view in the 4+1 view model. However, the conceptual architecture view is more conceptual and broader in scope. It takes into account existing software and hardware integration issues. This view is closely tied to the application domain. The functionality of the system is mapped to architectural elements called *conceptual components*. These conceptual components are not mapped directly to hardware, but they are mapped to a function that the system performs. This view provides an overview of the software architecture. It is the first place that people will go to find out how the system does what it is supposed to do. The concerns addressed in the conceptual view include the following:

1. How does the system fulfill the requirements?
2. How are the commercial off-the-shelf (COTS) components to be integrated, and how do they interact (at a functional level) with the rest of the system?
3. How are domain-specific hardware and/or software incorporated into the system?
4. How is functionality partitioned into product releases?
5. How does the system incorporate portions of prior generations of the product, and how will it support future generations?
6. How are product lines supported?
7. How can the impact of changes in the requirements domain be minimized?

Module View

The module view of software architecture shows how the elements of the software are mapped into modules and subsystems. The module architecture view shows how the system will be partitioned into separate run-time components. Some of the questions that are answered by this view are as follows:

1. How is the product mapped to the software platform?
2. What system support/services does it use and where?
3. How can testing be supported?
4. How can dependencies between modules be minimized?
5. How can reuse of modules and subsystems be maximized?
6. What techniques can be used to insulate the product from changes in third-party software, changes in the software platform, or changes to standards?

Execution View

The execution view of software architecture shows how modules are mapped onto the hardware of the system. This view shows the run-time entities and their attributes and how these entities are assigned to machines and

networks. It shows the memory usage, processor usage, and network usage expectations. This view shows the flow of control between hardware systems that execute the software of the system. Some questions that this view answers are the following:

1. How does the system meet its performance, recovery, and reconfiguration requirements?
2. How is resource usage balanced?
3. How are concurrency, replication, and distribution handled in the system?
4. How can the impact of changes in the run-time platform be minimized?

Code View

The code view shows how the source code and configuration are organized into packages, including package dependencies. It also shows how the executable portions of the system are built from the source code and configuration. This view answers the following questions:

1. How are the builds done? How long do they take?
2. How are versions and releases managed?
3. What tools are needed to support the development and configuration management environments?
4. How are integration and testing supported?

How important is it to define your architecture using one of these standard sets of views? The answer is: not at all. These views are presented here to give you an idea of the kinds of questions that need to be answered when you define your software architecture. When you are on a big project and you don't have a lot of experience with software architecture, it might be a good idea to download the 4+1 view model or purchase *Applied Software Architecture* (Hofmeister, Nord, and Soni 1999) book to get started. Both provide appropriate guidelines for the kinds of things to keep in mind, but every project and every organization are different. The best way to develop sound software architectures is to have the best software architects and developers. People who understand the intricacies of the technology, processes, and politics of your organization and who can apply some of the fundamental concepts outlined in this chapter will succeed in creating the best possible software architectures. If the method for capturing the architecture is completely ad-hoc or loosely based on the above methods, that is fine. Again, the most important issue is that the important questions get answered and that everyone knows what the answers are. What is not important is the particular format or style of the answers.

Architectural Styles, Patterns, and Metaphors

What is the difference between an architectural style, an architectural pattern, and a system metaphor? The answer is not much. An architectural style (Bass et al. 1997) and an architectural pattern (Buschmann et al. 1996) are essentially synonymous. A system metaphor is similar but more conceptual than an architecture pattern or an architectural style, and it relates more to a real-world concept than to a software engineering concept. An architectural style and an architectural pattern are similar to a design pattern in that they both describe a solution to a problem in a particular context. The only difference is the granularity at which they describe the solution. In a design pattern, the solution is relatively fine grained and is depicted at the level of language classes. In an architectural pattern, the solution is coarser grained and is described at the level of subsystems or modules and their relationships and collaborations. Each subsystem or module within an architectural pattern consists of many language classes that are designed using design patterns.

Every system needs a central, organizing concept. The conceptual integrity of the system depends on how strong this organizing concept is for the system. So what is an organizing concept? Some examples of architectural styles and patterns (Buschmann et al. 1996) include the following:

1. Model-View-Controller (MVC) architecture pattern. The MVC architectural style is a popular organizing concept for systems. The model represents the data for the system, the view represents the way the data are presented to the user, and the controller handles the logic for the system.
2. Publish-subscribe. The publish-subscribe architecture pattern is a system in which a publisher publishes data on a bus. Subscribers subscribe to portions of the data that are published by publishers on the bus. They register for various topics. When a message appears on the bus that matches the topic in which a subscriber is interested, the bus notifies the subscriber. The subscriber can then read the message from the bus.
3. Pipes and filters. Anyone who is familiar with UNIX systems will recognize this architectural pattern. The pipe and filter pattern allows a system to be assembled from small programs called *filters*. A filter has an input and an output. The filters are assembled into a chain in which each filter gets data from the previous filter in the chain, processes the data, and passes the data to the next filter in the chain. The best example of a pipe and filter system is a compiler. The lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and optimization occur in a pipe and filter style chain.
4. Layers. A layered system is one in which different layers of the system take care of a specific function of the system. Each layer in a layered architectural style is a package of software that has a well-defined interface and a few well-known dependencies with other layers. Each layer implements one technical function within the application. For example, a data access layer is responsible for encapsulating the technical means for accessing a database.

All data access requests to a database go through the data access layer for that database. The data access layer has the responsibility of hiding the data access mechanism from upstream layers. In a closed layer system, a layer may only access adjacent layers. In an open layer system, layers may access any other layer in the system, not just the ones to which they are adjacent.

Many other popular architectural patterns are used to provide a conceptual basis for software architecture. An application can use one or more architectural patterns in its development. Different subsystems can be built using different patterns. Also, the methods for component interaction can follow one pattern, while the internal implementations of the components could use a different pattern.

Architectural patterns are meant for software people to use and understand. One must study the pattern and understand the concept. MVC or pipes and filters have little or no basis in real life. The difference between an architectural pattern and a system metaphor is that a system metaphor is understandable by software people and customers alike. For example, a publish-subscribe pattern is also a system metaphor. In a publish-subscribe system, a publisher provides the data like a newspaper publisher or a book publisher. Each newspaper article or book relates to a particular topic. Consumers subscribe to various topics. When a magazine article or a book in which subscribers are interested comes out, they buy the book or magazine article. As you can imagine, system metaphors are not always pure. Magazine subscribers subscribe to the entire magazine, not just articles within the magazine. In addition, consumers don't subscribe to books, they purchase them. A system metaphor usually starts out with "The system is designed like a . . ."

Many products and concepts in software development come from metaphors. Following are a few examples:

- Library
- Stack
- Queue
- Desktop
- Account
- Directory
- Window
- Scheduler
- Dispatcher

A system can have many metaphors, but the metaphors have to make sense within the system to maintain its conceptual integrity. For example, Microsoft Windows has a desktop that contains windows. Would it have been better for windows to be placed on a wall? Are windows meant to move around on the wall? By combining multiple metaphors, some aspects of the metaphors must be forgotten and other must be emphasized. With time, the system itself can become a metaphor. We no longer think about the Windows desktop relating to a physical desktop or a window relating to an

actual window. Products can become metaphors in themselves. For example, a new system could say that the user interface is like the Windows desktop.

Conclusion

There is no magic when it comes to creating software architecture. Software architecture captures what is important about the design and communicates those concepts to the stakeholders on the project. Enterprise architecture is in many ways a product of the combined software architectures of the systems in the organization. Therefore, it is important to get the architecture right. To be credible, the architecture should be thin, approachable, readable, and understandable. With an intense focus on the architecture baseline and a quality attribute-based analysis of the software architecture as it is being built, it is likely that the architecture will suit the needs of the project.

In Chapter 4, we look at software product lines. Software product lines will allow an organization to leverage great architecture across multiple projects that have similar characteristics. Using great architecture and a product-line approach to delivering it, an organization can achieve great enterprise architecture.