Chapter 1
Understanding Software Architecture

1.1 What is Software Architecture?

The last 15 years have seen a tremendous rise in the prominence of a software engineering subdiscipline known as software architecture. *Technical Architect* and *Chief Architect* are job titles that now abound in the software industry. There's an International Association of Software Architects,[1] and even a certain well-known wealthiest geek on earth used to have "architect" in his job title in his prime. It can't be a bad gig, then?

I have a sneaking suspicion that "architecture" is one of the most overused and least understood terms in professional software development circles. I hear it regularly misused in such diverse forums as project reviews and discussions, academic paper presentations at conferences and product pitches. You know a term is gradually becoming vacuous when it becomes part of the vernacular of the software industry sales force.

This book is about software architecture. In particular it's about the key design and technology issues to consider when building server-side systems that process multiple, simultaneous requests from users and/or other software systems. Its aim is to concisely describe the essential elements of knowledge and key skills that are required to be a software architect in the software and information technology (IT) industry. Conciseness is a key objective. For this reason, by no means everything an architect needs to know will be covered. If you want or need to know more, each chapter will point you to additional worthy and useful resources that can lead to far greater illumination.

So, without further ado, let's try and figure out what, at least I think, software architecture really is, and importantly, isn't. The remainder of this chapter will address this question, as well as briefly introducing the major tasks of an architect, and the relationship between architecture and technology in IT applications.

[1] http://www.iasahome.org/web/home/home

1.2 Definitions of Software Architecture

Trying to define a term such as software architecture is always a potentially dangerous activity. There really is no widely accepted definition by the industry. To understand the diversity in views, have a browse through the list maintained by the Software Engineering Institute.[2] There's a lot. Reading these reminds me of an anonymous quote I heard on a satirical radio program recently, which went something along the lines of "the reason academic debate is so vigorous is that there is so little at stake".

I've no intention of adding to this debate. Instead, let's examine three definitions. As an IEEE member, I of course naturally start with the definition adopted by my professional body:

> Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.
>
> [ANSI/IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*]

This lays the foundations for an understanding of the discipline. Architecture captures system structure in terms of components and how they interact. It also defines system-wide design rules and considers how a system may change.

Next, it's always worth getting the latest perspective from some of the leading thinkers in the field.

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
>
> [L.Bass, P.Clements, R.Kazman, *Software Architecture in Practice (2nd edition),* Addison-Wesley 2003]

This builds somewhat on the above ANSI/IEEE definition, especially as it makes the role of abstraction (i.e., externally visible properties) in an architecture and multiple architecture views (structures of the system) explicit. Compare this with another, from Garlan and Shaw's early influential work:

> [Software architecture goes] beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.
>
> [D. Garlan, M. Shaw, *An Introduction to Software Architecture,* Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific, 1993]

It's interesting to look at these, as there is much commonality. I include the third mainly as it's again explicit about certain issues, such as scalability and

[2] http://www.sei.cmu.edu/architecture/definitions.html

distribution, which are implicit in the first two. Regardless, analyzing these a little makes it possible to draw out some of the fundamental characteristics of software architectures. These, along with some key approaches, are described below.

1.2.1 Architecture Defines Structure

Much of an architect's time is concerned with how to sensibly partition an application into a set of interrelated components, modules, objects or whatever unit of software partitioning works for you.[3] Different application requirements and constraints will define the precise meaning of "sensibly" in the previous sentence – an architecture must be designed to meet the specific requirements and constraints of the application it is intended for.

For example, a requirement for an information management system may be that the application is distributed across multiple sites, and a constraint is that certain functionality and data must reside at each site. Or, an application's functionality must be accessible from a web browser. All these impose some structural constraints (site-specific, web server hosted), and simultaneously open up avenues for considerable design creativity in partitioning functionality across a collection of related components.

In partitioning an application, the architect assigns responsibilities to each constituent component. These responsibilities define the tasks a component can be relied upon to perform within the application. In this manner, each component plays a specific role in the application, and the overall component ensemble that comprises the architecture collaborates to provide the required functionality.

Responsibility-driven design (see *Wirfs-Brock* in Further Reading) is a technique from object-orientation that can be used effectively to help define the key components in an architecture. It provides a method based on informal tools and techniques that emphasize behavioral modeling using objects, responsibilities and collaborations. I've found this extremely helpful in past projects for structuring components at an architectural level.

A key structural issue for nearly all applications is minimizing dependencies between components, creating a loosely coupled architecture from a set of highly cohesive components. A dependency exists between components when a change in one potentially forces a change in others. By eliminating unnecessary dependencies, changes are localized and do not propagate throughout an architecture (see Fig. 1.1).

[3]*Component* here and in the remainder of this book is used very loosely to mean a recognizable "chunk" of software, and not in the sense of the more strict definition in *Szyperski C. (1998) Component Software: Beyond Object-Oriented Programming, Addison-Wesley*

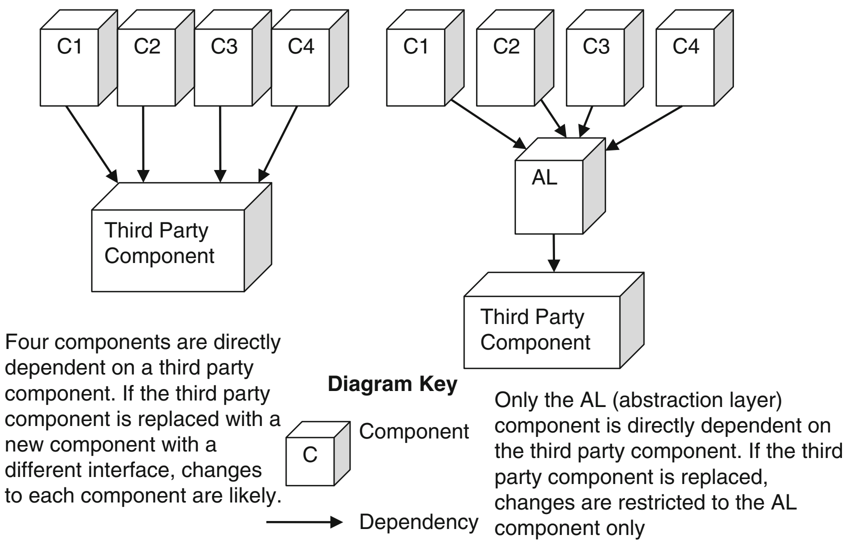
C1
C2
C3
C4
Third Party Component
Four components are directly dependent on a third party component. If the third party component is replaced with a new component with a different interface, changes to each component are likely.
C1
C2
C3
C4
AL
Third Party Component
Diagram Key
C
Component
Dependency
Only the AL (abstraction layer) component is directly dependent on the third party component. If the third party component is replaced, changes are restricted to the AL component only

Fig. 1.1 Two examples of component dependencies

Excessive dependencies are simply a bad thing. They make it difficult to make changes to systems, more expensive to test changes, they increase build times, and they make concurrent, team-based development harder.

1.2.2 Architecture Specifies Component Communication

When an application is divided into a set of components, it becomes necessary to think about how these components communicate data and control information. The components in an application may exist in the same address space, and communicate via straightforward method calls. They may execute in different threads or processes, and communicate through synchronization mechanisms. Or multiple components may need to be simultaneously informed when an event occurs in the application's environment. There are many possibilities.

A body of work known collectively as architectural patterns or styles[4] has catalogued a number of successfully used structures that facilitate certain kinds of component communication [see *Patterns* in Further Reading]. These patterns are essentially reusable architectural blueprints that describe the structure and interaction between collections of participating components.

Each pattern has well-known characteristics that make it appropriate to use in satisfying particular types of requirements. For example, the client–server pattern

[4]Patterns and styles are essentially the same thing, but as a leading software architecture author told me recently, "the patterns people won". This book will therefore use patterns instead of styles!

has several useful characteristics, such as synchronous request–reply communications from client to server, and servers supporting one or more clients through a published interface. Optionally, clients may establish sessions with servers, which may maintain state about their connected clients. Client–server architectures must also provide a mechanism for clients to locate servers, handle errors, and optionally provide security on server access. All these issues are addressed in the client–server architecture pattern.

The power of architecture patterns stems from their utility, and ability to convey design information. Patterns are proven to work. If used appropriately in an architecture, you leverage existing design knowledge by using patterns.

Large systems tend to use multiple patterns, combined in ways that satisfy the architecture requirements. When an architecture is based around patterns, it also becomes easy for team members to understand a design, as the pattern infers component structure, communications and abstract mechanisms that must be provided. When someone tells me their system is based on a three-tier client–server architecture, I know immediately a considerable amount about their design. This is a very powerful communication mechanism indeed.

1.3 Architecture Addresses Nonfunctional Requirements

Nonfunctional requirements are the ones that don't appear in use cases. Rather than define *what* the application does, they are concerned with *how* the application provides the required functionality.

There are three distinct areas of nonfunctional requirements:

- *Technical constraints*: These will be familiar to everyone. They constrain design options by specifying certain technologies that the application must use. "We only have Java developers, so we must develop in Java". "The existing database runs on Windows XP only". These are usually nonnegotiable.
- *Business constraints*: These too constraint design options, but for business, not technical reasons. For example, "In order to widen our potential customer base, we must interface with XYZ tools". Another example is "The supplier of our middleware has raised prices prohibitively, so we're moving to an open source version". Most of the time, these too are nonnegotiable.
- *Quality attributes*: These define an application's requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor. Chapter 3 discusses quality attributes in some detail.

An application architecture must therefore explicitly address these aspects of the design. Architects need to understand the functional requirements, and create a platform that supports these and simultaneously satisfies the nonfunctional requirements.

1.3.1 Architecture Is an Abstraction

One of the most useful, but often nonexistent, descriptions from an architectural perspective is something that is colloquially known as a *marketecture*. This is one page, typically informal depiction of the system's structure and interactions. It shows the major components and their relationships and has a few well-chosen labels and text boxes that portray the design philosophies embodied in the architecture. A *marketecture* is an excellent vehicle for facilitating discussion by stakeholders during design, build, review, and of course the sales process. It's easy to understand and explain and serves as a starting point for deeper analysis.

A thoughtfully crafted *marketecture* is particularly useful because it is an abstract description of the system. In reality, any architectural description must employ abstraction in order to be understandable by the team members and project stakeholders. This means that unnecessary details are suppressed or ignored in order to focus attention and analysis on the salient architectural issues. This is typically done by describing the components in the architecture as black boxes, specifying only their *externally visible properties*. Of course, describing system structure and behavior as collections of communicating black box abstractions is normal for practitioners who use object-oriented design techniques.

One of the most powerful mechanisms for describing an architecture is hierarchical decomposition. Components that appear in one level of description are decomposed in more detail in accompanying design documentation. As an example, Fig. 1.2 depicts a very simple two-level hierarchy using an informal notation, with two of the components in the top-level diagram decomposed further.

Different levels of description in the hierarchy tend to be of interest to different developers in a project. In Fig. 1.2, it's likely that the three components in the top-level description will be designed and built by different teams working on the

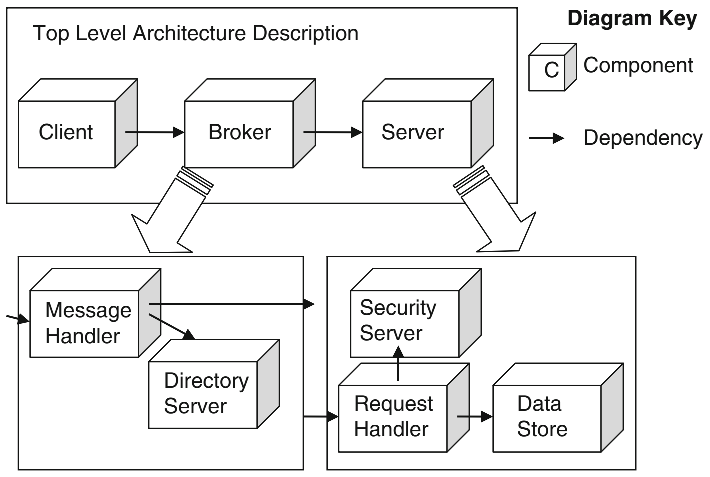

Fig. 1.2 Describing an architecture hierarchically

application. The architecture clearly partitions the responsibilities of each team, defining the dependencies between them.

In this hypothetical example, the architect has refined the design of two of the components, presumably because some nonfunctional requirements dictate that further definition is necessary. Perhaps an existing security service must be used, or the *Broker* must provide a specific message routing function requiring a directory service that has a known level of request throughput. Regardless, this further refinement creates a structure that defines and constrains the detailed design of these components.

The simple architecture in Fig. 1.2 doesn't decompose the *Client* component. This is, again presumably, because the internal structure and behavior of the client is not significant in achieving the application's overall nonfunctional requirements. How the *Client* gets the information that is sent to the *Broker* is not an issue that concerns the architect, and consequently the detailed design is left open to the component's development team. Of course, the *Client* component could possibly be the most complex in the application. It might have an internal architecture defined by its design team, which meets specific quality goals for the *Client* component. These are, however, localized concerns. It's not necessary for the architect to complicate the application architecture with such issues, as they can be safely left to the *Client* design team to resolve. This is an example of suppressing unnecessary details in the architecture.

1.3.2 Architecture Views

A software architecture represents a complex design artifact. Not surprisingly then, like most complex artifacts, there are a number of ways of looking at and understanding an architecture. The term "architecture views" rose to prominence in Philippe Krutchen's 1995[5] paper on the *4+1 View Model*. This presented a way of describing and understanding an architecture based on the following four views:

- *Logical view*: This describes the architecturally significant elements of the architecture and the relationships between them. The logical view essentially captures the structure of the application using class diagrams or equivalents.
- *Process view*: This focuses on describing the concurrency and communications elements of an architecture. In IT applications, the main concerns are describing multithreaded or replicated components, and the synchronous or asynchronous communication mechanisms used.
- *Physical view*: This depicts how the major processes and components are mapped on to the applications hardware. It might show, for example, how the database and web servers for an application are distributed across a number of server machines.

[5]P.Krutchen, *Architectural Blueprints–The "4+1" View Model of Software Architecture*, IEEE Software, 12(6) Nov. 1995.

- *Development view*: This captures the internal organization of the software components, typically as they are held in a development environment or configuration management tool. For example, the depiction of a nested package and class hierarchy for a Java application would represent the development view of an architecture.

These views are tied together by the architecturally significant use cases (often called scenarios). These basically capture the requirements for the architecture and hence are related to more than one particular view. By working through the steps in a particular use case, the architecture can be "tested", by explaining how the design elements in the architecture respond to the behavior required in the use case. We'll explore how to do this "architecture testing" in Chap. 5.

Since Krutchen's paper, there's been much thinking, experience, and development in the area of architecture views. Mostly notably is the work from the SEI, colloquially known as the "Views and Beyond" approach (see Further Reading). This recommends capturing an architecture model using three different views:

- *Module*: This is a structural view of the architecture, comprising the code modules such as classes, packages, and subsystems in the design. It also captures module decomposition, inheritance, associations, and aggregations.
- *Component and connector*: This view describes the behavioral aspects of the architecture. Components are typically objects, threads, or processes, and the connectors describe how the components interact. Common connectors are sockets, middleware like CORBA or shared memory.
- *Allocation*: This view shows how the processes in the architecture are mapped to hardware, and how they communicate using networks and/or databases. It also captures a view of the source code in the configuration management systems, and who in the development group has responsibility for each modules.

The terminology used in "Views and Beyond" is strongly influenced by the architecture description language (ADL) research community. This community has been influential in the world of software architecture but has had limited impact on mainstream information technology. So while this book will concentrate on two of these views, we'll refer to them as the structural view and the behavioral view. Discerning readers should be able to work out the mapping between terminologies!

1.4 What Does a Software Architect Do?

The environment that a software architect works in tends to define their exact roles and responsibilities. A good general description of the architect's role is maintained by the SEI on their web site.[6] Instead of summarizing this, I'll briefly describe, in no

[6]http://www.sei.cmu.edu/ata/arch_duties.html

particular order, four essential skills for a software architect, regardless of their professional environment.

- *Liaison*: Architects play many liaison roles. They liaise between the customers or clients of the application and the technical team, often in conjunction with the business and requirements analysts. They liaise between the various engineering teams on a project, as the architecture is central to each of these. They liaise with management, justifying designs, decisions and costs. They liaise with the sales force, to help promote a system to potential purchasers or investors. Much of the time, this liaison takes the form of simply translating and explaining different terminology between different stakeholders.
- *Software Engineering*: Excellent design skills are what get a software engineer to the position of architect. They are an essential prerequisite for the role. More broadly though, architects must promote good software engineering practices. Their designs must be adequately documented and communicated and their plans must be explicit and justified. They must understand the downstream impact of their decisions, working appropriately with the application testing, documentation and release teams.
- *Technology Knowledge*: Architects have a deep understanding of the technology domains that are relevant to the types of applications they work on. They are influential in evaluating and choosing third party components and technologies. They track technology developments, and understand how new standards, features and products might be usefully exploited in their projects. Just as importantly, good architects know what they don't know, and ask others with greater expertise when they need information.
- *Risk Management*: Good architects tend to be cautious. They are constantly enumerating and evaluating the risks associated with the design and technology choices they make. They document and manage these risks in conjunction with project sponsors and management. They develop and instigate risk mitigation strategies, and communicate these to the relevant engineering teams. They try to make sure no unexpected disasters occur.

Look for these skills in the architects you work with or hire. Architects play a central role in software development, and must be multiskilled in software engineering, technology, management and communications.

1.5 Architectures and Technologies

Architects must make design decisions early in a project lifecycle. Many of these are difficult, if not impossible, to validate and test until parts of the system are actually built. Judicious prototyping of key architectural components can help increase confidence in a design approach, but sometimes it's still hard to be certain of the success of a particular design choice in a given application context.

Due to the difficulty of validating early design decisions, architects sensibly rely on tried and tested approaches for solving certain classes of problems. This is one of the great values of architectural patterns. They enable architects to reduce risk by leveraging successful designs with known engineering attributes.

Patterns are an abstract representation of an architecture, in the sense that they can be realized in multiple concrete forms. For example, the publish–subscribe architecture pattern describes an abstract mechanism for loosely coupled, many-to-many communications between publishers of messages and subscribers who wish to receive messages. It doesn't however specify how publications and subscriptions are managed, what communication protocols are used, what types of messages can be sent, and so on. These are all considered implementation details.

Unfortunately, despite the misguided views of a number of computer science academics, abstract descriptions of architectures don't yet execute on computers, either directly or through rigorous transformation. Until they do, abstract architectures must be reified by software engineers as concrete software implementations.

Fortunately, the software industry has come to the rescue. Widely utilized architectural patterns are supported in a variety of prebuilt frameworks available as commercial and open source technologies. For a matter of convenience, I'll refer to these collectively as commercial-off-the-shelf (COTS) technologies, even though it's strictly not appropriate as many open source products of very high quality can be freely used (often with a *pay-for-support* model for serious application deployments).

Anyway, if a design calls for publish–subscribe messaging, or a message broker, or a three-tier architecture, then the choices of available technology are many and varied indeed. This is an example of software technologies providing reusable, application-independent software infrastructures that implement proven architectural approaches.

As Fig. 1.3 depicts, several classes of COTS technologies are used in practice to provide packaged implementations of architectural patterns for use in IT systems. Within each class, competing commercial and open source products exist. Although these products are superficially similar, they will have differing feature sets, be implemented differently and have varying constraints on their use.

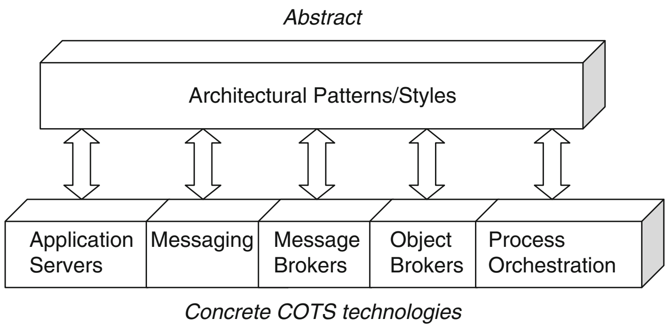

Fig. 1.3 Mapping between logical architectural patterns and concrete technologies

Architects are somewhat simultaneously blessed and cursed with this diversity of product choice. Competition between product vendors drives innovation, better feature sets and implementations, and lower prices, but it also places a burden on the architect to select a product that has quality attributes that satisfy the application requirements. All applications are different in some ways, and there is rarely, if ever, a *one-size-fits-all* product match. Different COTS technology implementations have different sets of strengths and weaknesses and costs, and consequently will be better suited to some types of applications than others.

The difficulty for architects is in understanding these strengths and weaknesses early in the development cycle for a project, and choosing an appropriate reification of the architectural patterns they need. Unfortunately, this is not an easy task, and the risks and costs associated with selecting an inappropriate technology are high. The history of the software industry is littered with poor choices and subsequent failed projects. To quote Eoin Woods,[7] and provide another extremely pragmatic definition of software architecture:

> Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.

Chapters 4–6 provide a detailed description and analysis of these infrastructural technologies.

1.6 Architect Title Soup

Scan the jobs advertisements. You'll see chief architects, product architects, technical architects, solution architects (I want to place a spoof advert for a problem architect), enterprise architects, and no doubt several others. Here's an attempt to give some general insights into what these mean:

- *Chief Architect*: Typically a senior position who manages a team of architects within an organization. Operates at a broad, often organizational level, and coordinates efforts across system, applications, and product lines. Very experienced, with a rare combination of deep technical and business knowledge.
- *Product/Technical/Solution Architect*: Typically someone who has progressed through the technical ranks and oversees the architectural design for a specific system or application. They have a deep knowledge of how some important piece of software really works.
- *Enterprise Architect*: Typically a much less technical, more business-focus role. Enterprise architects use various business methods and tools to understand, document, and plan the structure of the major systems in an enterprise.

The content of this book is relevant to the first two bullets above, which require a strong computer science background. However, enterprise architects are somewhat

[7] http://www.eoinwoods.info/

different beasts. This all gets very confusing, especially when you're a software architect working on enterprise systems.

Essentially, enterprise architects create documents, roadmaps, and models that describe the logical organization of business strategies, metrics, business capabilities, business processes, information resources, business systems, and networking infrastructure within the enterprise.[8] They use frameworks to organize all these documents and models, with the most popular ones being TOGAF[9] and the Zachman Framework.[10]

Now if I'm honest, the above pretty much captures all I know about enterprise architecture, despite having been involved for a short time on an enterprise architecture effort! I'm a geek at heart, and I have never seen any need for computer science and software engineering knowledge in enterprise architecture. Most enterprise architects I know have business or information systems degrees. They are concerned with how to "align IT strategy and planning with company's business goals", "develop policies, standards, and guidelines for IT selection", and "determine governance". All very lofty and important concerns, and I don't mean to be disparaging, but these are not my core interests. The tasks of an enterprise architect certainly don't rely on a few decades of accumulated computer science and software engineering theory and practice.

If you're curious about enterprise architecture, there are some good references at the end of this chapter. Enjoy.

1.7 Summary

Software architecture is a fairly well defined and understood design discipline. However, just because we know what it is and more or less what needs doing, this doesn't mean it's mechanical or easy. Designing and evaluating an architecture for a complex system is a creative exercise, requiring considerable knowledge, experience and discipline. The difficulties are exacerbated by the early lifecycle nature of much of the work of an architect. To my mind, the following quote from Philippe Krutchen sums up an architect's role perfectly:

> The life of a software architect is a long (and sometimes painful) succession of sub-optimal decisions made partly in the dark

The remainder of this book will describe methods and techniques that can help you to shed at least some light on architectural design decisions. Much of this light comes from understanding and leveraging design principles and technologies that have proven to work in the past. Armed with this knowledge, you'll be able to

[8]http://en.wikipedia.org/wiki/Enterprise_Architecture

[9]http://www.opengroup.org/togaf/

[10]http://www.zachmaninternational.com/index.php/the-zachman-framework

tackle complex architecture problems with more confidence, and after a while, perhaps even a little panache.

1.8 Further Reading

There are lots of good books, reports, and papers available in the software architecture world. Below are some I'd especially recommend. These expand on the information and messages covered in this chapter.

1.8.1 General Architecture

In terms of defining the landscape of software architecture and describing their project experiences, mostly with defense projects, it's difficult to go past the following books from members of the Software Engineering Institute.

L. Bass, P. Clements, R Kazman. *Software Architecture in Practice*, Second Edition. Addison-Wesley, 2003.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond.* 2nd Edition, Addison-Wesley, 2010.

P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, 2002.

For a description of the "Decomposition Style", see *Documenting Software Architecture*, page 53. And for an excellent discussion of the *uses* relationship and its implications, see the same book, page 68.

The following are also well worth a read:

Nick Rozanski, Eion Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley 2005

Richard N. Taylor, Nenad Medvidovic, Eric Dashofy, *Software Architecture: Foundations, Theory, and Practice,* John Wiley and Sons, 2009

Martin Fowler's article on the role of an architect is an interesting read.

Martin Fowler, *Who needs an Architect?* IEEE Software, July-August 2003.

1.8.2 Architecture Requirements

The original book describing use cases is:

I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

Responsibility-driven design is an incredibly useful technique for allocating functionality to components and subsystems in an architecture. The following should be compulsory reading for architects.

R. Wirfs-Brock, A. McKean. Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, 2002.

1.8.3 Architecture Patterns

There's a number of fine books on architecture patterns. Buschmann's work is an excellent introduction.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal,. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, 1996.

D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects.* John Wiley & Sons, 2000.

Two recent books that focus more on patterns for enterprise systems, especially enterprise application integrations, are well worth a read.

M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

G. Hohpe, B. Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003.

1.8.4 Technology Comparisons

A number of papers that emerged from the Middleware Technology Evaluation (MTE) project give a good introduction into the issues and complexities of technology comparisons.

P. Tran, J. Gosper, I. Gorton. *Evaluating the Sustained Performance of COTS-based Messaging Systems.* in Software Testing, Verification and Reliability, vol 13, pp 229–240, Wiley and Sons, 2003.

I. Gorton, A. Liu. *Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications,* in IEEE Internet Computing, vol.7, no. 3, pages 18–23, 2003.

A. Liu, I. Gorton. *Accelerating COTS Middleware Technology Acquisition: the i-MATE Process.* in IEEE Software, pages 72–79,volume 20, no. 2, March/April 2003.

1.8.5 Enterprise Architecture

In my humble opinion, there's some seriously shallow books written about enterprise architecture. I survived through major parts of this book, so would recommend it as a starting point.

James McGovern, Scott Ambler, Michael Stevens, James Linn, Elias Jo and Vikas Sharan, *The Practical Guide to Enterprise Architecture*, Addison-Wesley, 2003.

Another good general, practical book is:

Marc Lankhorst, Enterprise Architecture at Work, Springer-Verlag, 2009

I'm sure there's joy to be had in the 700+ pages of the latest *TOGAF version 9.0* book (Van Haren publishing, ISBN: 9789087532307), but like Joyce's *Ulysses*, I suspect it's a joy I will never have the patience to savor. If the Zachman Framework is more your thing, there's a couple of *ebooks*, which look informative at a glance:

http://www.zachmaninternational.com/index.php/ea-articles/25-editions