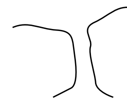
A Software Architecture Primer

John Reekie
Rohan McAdam



Angophora Press

This is a sample chapter (or chapters) from A Software Architecture Primer, by John Reekie and Rohan McAdam.

ISBN 0-646-45841-8

Copyright © 2006 H. John Reekie and Rohan J. McAdam

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the authors.

Contents

Preface **ix**

Acknowledgments **xi**

1 An Introduction to Architecture **1**
1.1 Software systems . . . 2
1.2 A simple definition of software architecture . . . 2
1.3 Understanding software architecture... . . . 3
1.3.1 By analogy: making buildings . . . 4
1.3.2 By its place in the development lifecycle . . . 4
1.3.3 By its impact on system lifetime . . . 5
1.3.4 By "-ilities" . . . 6
1.3.5 By its role in a software business . . . 7
1.3.6 By example . . . 8
1.3.7 As a conceptual framework . . . 9
1.4 The rest of this book . . . 10

2 Architectural Analysis **11**
2.1 The architectural response . . . 12
2.2 Stakeholder needs . . . 13
2.3 Contextual factors . . . 16
2.4 Functional requirements . . . 19
2.5 Non-functional requirements . . . 22
2.5.1 Runtime quality attributes . . . 22
2.5.2 Non-runtime quality attributes . . . 24

3 Architectural Design **27**
3.1 Architectural structure . . . 28
3.2 Architectural behavior . . . 29
3.3 System architecture . . . 33
3.4 Models . . . 36
3.5 Views . . . 37

4 Conceptual Architecture **39**
4.1 The elements of conceptual architecture 40
4.2 Conceptual architecture design 43
4.2.1 The initial conceptual architecture 43
4.2.2 Refining the architecture 47
4.2.3 Initial behavioral exploration 48
4.2.4 Component stereotypes 50
4.2.5 Data models 53
4.3 Conceptual architecture gotcha's 55

5 Execution Architecture **59**
5.1 The elements of execution architecture 60
5.2 Execution architecture design 62
5.2.1 Concurrent subsystems 62
5.2.2 Process models 64
5.2.3 Execution stereotypes 65
5.2.4 Detailed concurrency models 67
5.2.5 Identifying concurrent components 69
5.3 Behavior 72
5.3.1 Binding execution and conceptual models 72
5.3.2 Runtime quality attributes 72
5.3.3 Dynamic concurrency 73
5.4 Deployment 76
5.4.1 Physical components 76
5.4.2 Deployment models 77

6 Implementation Architecture **81**
6.1 The elements of implementation architecture 82
6.2 Implementation architecture design 86
6.2.1 Finding application components 86
6.2.2 Finding infrastructure components 87
6.2.3 Interface design 88
6.3 Behavior 90
6.3.1 Sequence diagrams 90
6.3.2 Non-runtime quality attributes 91
6.4 Containers and component frameworks 93
6.5 Architectural prototyping 95

7 Architectural Styles **99**
7.1 Abstraction layers 100
7.1.1 Characteristics 100
7.1.2 Behavior 103
7.1.3 Variations 104
7.2 Pipe-and-filter architectures 105
7.2.1 Characteristics 106
7.2.2 Variations 107

7.2.3 Static scheduling . . . 108
7.3 N-tier architectures . . . 110
7.3.1 Characteristics . . . 111
7.3.2 Variations . . . 112
7.4 Notification architectures . . . 116
7.5 On the application of style . . . 117

8 Quality Attributes 119
8.1 Performance . . . 120
8.2 Usability . . . 121
8.3 Reliability and testability . . . 125
8.4 Security . . . 127
8.5 Configurability and scalability . . . 131
8.6 Maintainability . . . 133

9 Architecture and Process 135
9.1 Architecture in the development lifecycle . . . 136
9.1.1 The Unified Process . . . 136
9.1.2 Agile development . . . 138
9.1.3 Other development styles . . . 140
9.2 Views and viewpoints . . . 142
9.2.1 IEEE 1471 . . . 143
9.2.2 Clements *et al* . . . 145
9.2.3 Rozanski and Woods . . . 149
9.2.4 The "4+1" model . . . 150

10 Industrial Systems 153
10.1 The industrial environment . . . 154
10.1.1 The physical environment . . . 154
10.1.2 The business environment . . . 154
10.2 The organization of industrial enterprises . . . 155
10.3 User needs and responses . . . 156
10.3.1 Control engineers . . . 156
10.3.2 Console operators . . . 159
10.3.3 Production planners . . . 163
10.4 Overall system structure . . . 165
10.4.1 The Purdue model . . . 165
10.4.2 Architectural styles . . . 166
10.5 Future challenges and opportunities . . . 169
10.5.1 Challenges . . . 169
10.5.2 Opportunities . . . 170
10.6 Conclusion . . . 171

Bibliography 173

Index 177

Chapter 1

An Introduction to Architecture

The field of software development is relatively young, and the sub-field of software architecture is younger still. Although many of the basic principles of what we now call "software architecture" have been known and practiced for decades, software architecture has only been recognized as a distinct practice for a little over ten years. Although Perry and Wolf published on software architecture as early as 1989 [Perry and Wolf, 1989], the publication of Shaw and Garlan's book, *Software Architecture: Perspectives on an Emerging Discipline* [Garlan and Shaw, 1996], could be considered to mark the starting point of software architecture as a discipline within the broader field of software development.

Since software architecture is so young, it is still defining itself. We will provide one possible definition of what "software architecture" is. Our definition concurs with most other published definitions... yet, we think there is more to software architecture than is (or perhaps can be) captured by such a short definition. So we will provide several other ways of understanding what software architecture *is*. In a similar vein to the now-accepted notion that the architecture of a system is expressed using multiple views, each showing something different about the system, our way of "defining" software architecture is to present several "views" of it!

But first, we must define what we mean by a "software system."

1.1 Software systems

A *system* is a collection of parts and a set of unifying principles or purposes that together makes the collection of parts form a unified whole with a single purpose. The system can itself form a part of another, larger, system.

For example, in biology, the cardio-vascular system transports blood throughout the body, carrying oxygen from the lungs to the bodily tissues, and removing waste products from those tissues. It consists of the heart, arteries and veins, and the blood itself. Related systems include the lymphatic system and the respiratory system.

A software system, then, is a system in which the parts, and the realization of the unifying principles (communication between the parts, for example) are mostly of software elements. There are other elements that come into play, such as the hardware on which the software runs, and the network hardware that various elements of the system will use to communicate. Still, a software system, as opposed to, say, a computer system, or a network topology, is primarily concerned with elements that are constructed by writing software. As in the biology example, software systems may themselves be parts of a larger system. In this case, we often use the term *subsystem*.

Now, since a system has a single purpose, so does a software system. Whether it is to route packets to the appropriate destination, or to manage the accounts of a large company, or something else, the system has to have some purpose. We will continue this discussion in the next section.

1.2 A simple definition of software architecture

The term "architecture" is applied in many fields. As a metaphor for building architecture (the first kind of architecture), practitioners in different disciplines have evolved different but similar understandings of what architecture means in that field. Examples include building architecture, from which the other fields borrow the term; naval architecture, the art and science of designing and building boats and ships; and computer architecture, the study of and discipline related to the structure and operation of computer processors.

Many things that we think of as "architecture" are often characterized as being both an "art" and a "science." We can also note that things that we think of as "architecture" have the following characteristics:

- The whole consists of smaller parts.
- The parts have relations to each other.
- When put together, the parts form a whole that has some designed purpose and fills a specific need.

The definition feels a bit redundant, but we think it works well. Where our definition is a little different from most such definitions is the addition of

the phrase "has some designed purpose and fills a specific need." We think that, if you are after a short definition, this phrase adds nicely to the primarily structural flavor of the rest of the definition. That is, if you are designing the architecture of a system, you do so in such a way that the system does what you've designed it to do, and it does *that* because there's a need—driven by human concerns—that it do so. Software architecture is no different, and we come back to this throughout the book: the software systems we build are there to fill a human need, and it is good for us technical types not to forget this. The cost of forgetting is that we build systems that either no-one wants, or no-one is prepared to pay for. And *that* is not a very rewarding way to spend a career in software.

The word "architecture" itself has several levels of meaning. We can consider the above definition to be describing:

1. The architecture of a software system

   When we build a software system, part of that process is designing the architecture. This architecture describes the system as a whole, breaks it down into parts, and then says how the parts come together to have a designed purpose and meet a human need.

2. A field of study

   This is what is usually meant by "architecture" in the building sense. Software architecture is the knowledge that exists about how to go about designing the architecture of software systems. This book, for example, is a book about software architecture in this sense. (It is not a book about the architecture of any particular software system.)

3. What software architects do

   This could almost be considered a definition on its own. What is software architecture? It's what software architects do. Trite, but in some ways, quite true. A software architect goes about designing a system to have a purpose and fill a need, by considering the whole, the parts, and then the whole again, and the way he or she goes about that is called "software architecture."

The last item, in particular, pushes the boundaries of what our definition above can encompass, because it turns out that software architects do a lot of things other than what the definition said. So, let's look at some other ways of "defining" software architecture.

1.3 Understanding software architecture...

In terms of understanding what "software architecture" is, it might be easier to explain in ways other than by providing definitions. In this section, we will provide ways of understanding what software architecture is by placing it into the context of a range of different software development activities.

1.3.1 By analogy: making buildings

Often, software architecture is likened to building architecture. A well-known analogy is that provided by Grady Booch, in which he examines software development on three scales: building a dog kennel, building a house, and building a skyscraper. Clearly, when building a dog kennel, a significant investment in architectural design is not required—in fact, if one were handy with building things out of wood, it may not even be necessary to make any plans other than the most cursory measurements (how big is the dog so we know what size the door needs to be) and plans (a quick "back of the envelope" calculation to be sure that we have enough wood).

When building a house, an "architect" may or may not be necessary. Almost certainly, plans of some kinds will be needed. But more than that—well, it depends on the nature of the dwelling, the needs of the owners, and the skill of the builders.

When building a skyscraper, however, architects are certainly needed, in order to create the vision, the plan, and the structure of the building. Attempting to build a skyscraper without a significant investment in architecture would of course be a complete disaster.

In software development, it is common to use analogy and metaphor to explain what it is that we, the builders of these software systems, think it is that we are doing. This is both for the benefit of ourselves, in order that we can form and define our own profession, and for the benefit of outsiders, in order that we might explain to them what it is that we do and why it is so difficult. We have to use analogy and metaphor, because the software itself is so intangible.

It is worth remembering, though, that analogies and metaphors are only that: analogies and metaphors. In their classic book *Metaphors We Live By* [Lakoff and Johnson, 1980], Lakoff and Johnson explain that metaphors illuminate as much by their differences and limits as they do by their similarities. Thus, the kennel/house/skyscraper analogy is meant to illustrate an important point, which is that the way that we approach a problem varies according to the scale of the problem. It does not mean (as it sometimes appears to be taken) that we are supposed to find the software equivalent of concrete-pourers and brick-layers. Software development is an extremely challenging intellectual task, and trivializing or glorifying any role in this complex co-operative venture is likely to be futile and counter-productive.

1.3.2 By its place in the development lifecycle

While architecture plays a role throughout the development of a software system, much of the architectural design work fits neatly between requirements and analysis, and the beginnings of design and implementation. Figure 1.1 illustrates the place of architecture in the Evolutionary Delivery lifecycle model, as explained by Steve McConnell [McConnell, 1996]. From the diagram, it is clear that in this lifecycle model, architectural design notionally follows requirements gathering and analysis, and that there is also an iterative feedback loop

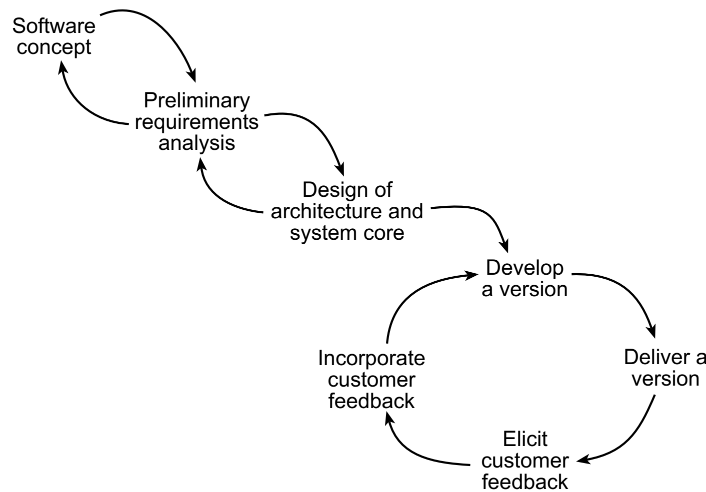


Figure 1.1: Architecture in the software development lifecycle

from architecture back to requirements analysis.

Evolutionary Delivery is an iterative lifecycle model, and the first development iteration can be used to develop the key architectural elements of the system. By the end of this iteration, key subsystems, communications infrastructure, external libraries and other packages, and system startup and shutdown mechanisms should be in place. In addition, static structures such as directory structures, regression testing infrastructure, and configuration management should also be in place.

In subsequent iterations, functionality is added to flesh out this "skeleton." In this particular lifecycle model, each iteration adds a pre-planned set of features or functionality, which is then reviewed by the development team and customers to fine-tune planning for the next iteration. Although not shown on the diagram, elements of the architecture may be altered as well, although this becomes increasingly expensive as the iteration count increases.

1.3.3 By its impact on system lifetime

Although "everybody knows" that maintenance consumes over half of the money put into any given system, software engineering texts still focus mostly on the development of the system, up until deployment. But if we consider the lifecycle of the *system itself*, rather than just the development lifecycle, we might get a diagram as in Figure 1.2.

Many of the issues that software architecture addresses impact the whole lifetime of the system. For example, architecture explicitly considers maintainability, or the ability of the system to be maintained. It addresses configurability, or the ability of the system to be configured to a particular installation and

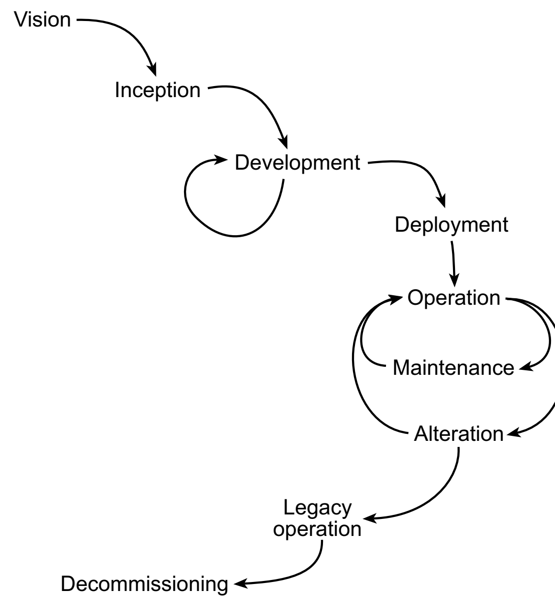


Figure 1.2: The lifecycle of a software system

hardware structure. It addresses scalability, or the ability of the system to have its capacity (throughput, number of users, and so on) increased when needed during its operational lifetime.

Many of the decisions made during the architectural design process will be trade-offs of these types of attributes, against more commonly-considered attributes such as performance and reliability. These decisions will be made based on the relative importance of factors such as the anticipated future use of the system, as well as on immediate needs such as feasibility, cost, and time to delivery.

1.3.4 By "-ilities"

Suppose you were given a functional specification of a software system, and then somehow implemented it. The system is acceptable, since it meets the functional specification, right?

Wrong. Software is a complex technology, and there are myriad ways of implementing that same functional specification. And each implementation will have different characteristics that will improve or detract from the overall "worth" of that system within the particular context in which it is developed and used. Two obvious examples are time to delivery and cost of development, which are important in most models of development (but not all). Let's look at a few more.

Suppose, for example, that your system worked perfectly, and processed 100

whatsits per second. Business improves and you now need to process 200 whatsits per second—what do you do? You cannot simply run it on two computers, since you didn't design it to be run that way. We've just run into two key system qualities—performance and scalability—that needed to be considered much earlier.

Suppose that once the system is running, an unexpected issue arises: a very particular combination of circumstances causes unexpected results. Well, clearly, the system doesn't meet its functional specification, as you thought. But more importantly, what are you going to do about it? You have to find the problem, and then fix it. A system that has had the attributes testability and maintainability explicitly considered during its earliest design phases is likely to be fixed in less time and with less cost than one that hasn't.

Attributes such as those just mentioned are sometimes called the "-ilities." We will call them quality attributes in this book. Sometimes, it is said that the purpose of software architecture is to consider the quality attributes. We think this statement might be a bit limiting, but the quality attributes certainly occupy much of the territory of software architecture.

1.3.5 By its role in a software business

The purpose of a software system and the specific needs that it fills are inevitably linked to the goals and aspirations of the organization developing the system. These goals are typically commercially driven, but may take on a more benevolent character in some cases. Regardless of the motivation, the form and structure of a software system is critical to the success of the system and hence the success of the organization developing the system. This places software architecture firmly within the business context of organizations developing software systems.

The business of developing software is a process of reconciling the concerns of a diverse range of stakeholders (customers, users, designers, developers, management, investors, and so on). Sitting as it does at the interface between what is needed (the requirements) and how those needs will be addressed (the software design), software architecture can provide a vehicle for communication between the various stakeholders.

The architectural vision of a system can be used to convince investors of the long-term relevance and viability of the system. The architectural design can be used to demonstrate to customers that the system will be able to perform its mission in the particular ways that matter to those customers (its "-ilities"). Users can gain an understanding of how they will interact with the system based on its architecture.

The architecture of a system also establishes the design framework for the software designers and developers whose job it is to implement the system. An implementation that fails to conform to the system's architecture will have repercussions that can be traced back to the concerns of the stakeholders for whom the system is being built.

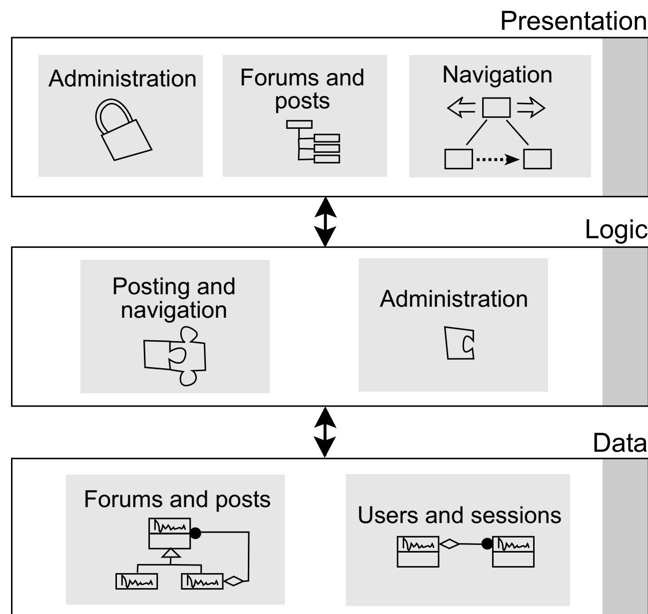


Figure 1.3: A web-based bulletin board

Software architecture has a central role in facilitating communication within this business context, and it is the software architects that carry the message. As a result, software architects enjoy a wide range of professional relationships with all of the key stakeholders in a software development enterprise.

1.3.6 By example

Software architecture can also be understood by example—that is, by looking at the architecture of different systems and comparing them. Figure 1.3 is a diagram that might appear in a description of the architecture of a web-based bulletin board, while Figure 1.4 is a diagram that might appear as part of the architecture documentation for a video-processing system. Clearly, the purpose of each system is very different, but so is its architecture.

The web-based system is structured in what is known as a "3-tier" architecture. Each of the three layers, called Presentation, Logic, and Data, has a different and distinct purpose. Respectively:

- Presentation of information to the user (that is, the user interface)
- Execution of the logic associated with the application
- Storage of persistent data—in this case, the bulletin board configuration and the posts made by its users

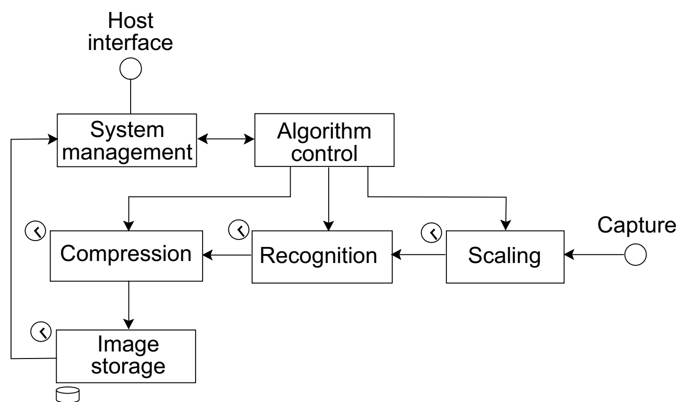


Figure 1.4: A video-processing system

Quality attributes that will be important for this system include response time (you generally don't want to wait more than a second or two for a web page), scalability (if the site is successful, many processors may be needed to support the traffic), and availability (since it's a web site, people expect it to be working all the time).

The video processing system, in contrast, has real-time requirements, since video is coming in through the *Capture* interface in real time. Each successive stage in the pipeline labeled *Scaling*, *Recognition*, and *Compression*, performs an independent processing step on each frame of video data. In this diagram, the little clock-like icon indicates a component with real-time requirements—in the system implementation, a dedicated scheduling component of some kind will be needed to ensure that frames are not dropped.

Quality attributes that will be important for this system will most likely be related to performance: that is, the system must be able to process all frames received at the given frame rate. Extensibility may be important, as it may be envisaged that better processing algorithms will become available in the future, so it must be relatively easy to add new processing modules.

Now, you could probably create a video-processing system with an architecture like Figure 1.3, and a web-based bulletin board with an architecture like Figure 1.4. The architecture, however, would be a very poor fit to the application and will be unlikely to realize the desired quality attributes.

1.3.7 As a conceptual framework

Finally, software architecture is a conceptual framework that helps us to understand and build a software system. In *The Mythical Man-Month* [Brooks, 1974], Fred Brooks pointed out the importance of the *conceptual integrity* of a system. A clean and clearly-defined architectural core contributes greatly to conceptual integrity. The overall structure and organization of the system, the methods

of communication between the different parts of the system, and the principles through which the architecture has been and should continue to develop, form this architectural core.

1.4 The rest of this book

Now that we have presented an overview of and the rationale for software architecture, we will move on to presenting ideas and techniques that you can apply in your own work. At first, these ideas and techniques may seem a little disconnected from each other, but once you become familiar with them, we think that you will find that they will mesh well together and with the system design and development techniques that you already use.

In Chapters 2 and 3, we capture, as quickly and easily as possible, key elements of the context in which the system is built: the system overview, constraints placed on the system, and stakeholder expectations. We then proceed to explore an initial design approach to the system's architecture, using simple "box and line" diagrams for describing the structure of a system. We end by tying the structure of the system to its dynamic behavior.

In Chapters 4 to 6, we look at three different "views" of a system. It's generally recognized that a complex system needs to be described in a number of different ways, each focusing on a different set of concerns. In these three chapters, we focus on the conceptual structure of the system, on its runtime structure, and on its build-time structure.

In Chapter 7, we look at large-scale patterns that occur in systems. These patterns or "styles" provide a vocabulary and design guidance for a number of commonly-occurring system structures, such as N-tier architectures, pipe-and-filter architectures, and so on.

Chapter 8 explores quality attributes in more detail. We select a few interesting quality attributes and explore how they interact with different types of system.

Chapter 9 discusses how architecture interacts with software development processes. Architecture can also be treated more formally than we have done in most of this book, and so we provide an introduction to the formal definition of views as given by the IEEE 1471 standard.

Chapter 10 looks a little more deeply at the place of software architecture in a particular type of software system: industrial control systems. This chapters places the concepts described in this book into the context of a specific application domain.

Bibliography

[Abrahamsson et al., 2002] Abrahamsson, P., Salo, O., Ronkainen, J., and Juhani, W. (2002). *Agile Software Development Methods: Review and Analysis.* VTT Publications.

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice.* Addison-Wesley. Second edition.

[Boehm and Turner, 2003] Boehm, B. and Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed.* Addison-Wesley.

[Bosch, 2000] Bosch, J. (2000). *Design and Use of Software Architectures.* Addison-Wesley.

[Brooks, 1974] Brooks, F. P. (1974). *The Mythical Man Month and Other Essays on Software Engineering.* Addison Wesley Longman Publishing Company. Republished by Addison-Wesley in 1995.

[Brown et al., 1998] Brown, W. J., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley.

[Buhr, 1998] Buhr, R. (1998). Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155.

[Buhr and Casselman, 1995] Buhr, R. J. A. and Casselman, R. S. (1995). *Use Case Maps for Object-Oriented Systems.* Prentice Hall. Available online at http://www.usecasemaps.org/.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons.

[Carroll, 1995] Carroll, J. M., editor (1995). *Scenario-Based Design: Envisioning Work and Technology in System Development.* John Wiley and Sons.

[Clements et al., 2003] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2003). *Documenting Software Architectures: Views and Beyond.* Addison-Wesley.

[Cockburn, 2000] Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley.

[Cockburn, 2001] Cockburn, A. (2001). *Agile Software Development*. Addison-Wesley.

[Constantine and Lockwood, 1999] Constantine, L. L. and Lockwood, L. A. (1999). *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley.

[Cooper, 1999] Cooper, A. (1999). *The Inmates Are Running the Asylum : Why High Tech Products Drive Us Crazy and How To Restore The Sanity*. Sams.

[Cusumano and Selby, 1997] Cusumano, M. A. and Selby, R. W. (1997). How Microsoft builds software. *Communications of the ACM*, 40(6):53–61.

[Foote and Yoder, 1997] Foote, B. and Yoder, J. W. (1997). Big Ball of Mud. In *Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97)*. Available electronically on http://www.laputan.org/.

[Gamma et al., 1995] Gamma, E., Helm, R., and Johnson, R. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[Garlan and Shaw, 1996] Garlan, D. and Shaw, M. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

[Grady and Caswell, 1987] Grady, R. B. and Caswell, D. L. (1987). *Software Metrics: Establishing a Company-wide Program*. Prentice Hall.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall International.

[Hofmeister et al., 2000] Hofmeister, C., Nord, R., and Soni, D. (2000). *Applied Software Architecture*. Addison-Wesley.

[IEEE, 2000] IEEE (2000). *IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. The Institution of Electrical and Electronics Engineers, Inc.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proceedings of International Federation for Information Processing Congress 74*, pages 471–475. North Holland Publishing Co.

[Kruchten, 1995] Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50.

[Lakoff and Johnson, 1980] Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By*. University of Chicago Press.

[McCarthy, 1995] McCarthy, J. (1995). *Dynamics of Software Development.* Microsoft Press.

[McConnell, 1996] McConnell, S. (1996). *Rapid Development.* Microsoft Press.

[Miller, 1968] Miller, R. B. (1968). Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference*, volume 33, pages 267–277. Summarized in [Nielsen, 1993].

[Nielsen, 1993] Nielsen, J. (1993). *Usability Engineering.* Academic Press.

[Norman, 1990] Norman, D. (1990). *The design of everyday things.* Doubleday Books. Re-issue edition.

[Open Source Initiative, 2006] Open Source Initiative (2006). History of the OSI. Online at http://www.opensource.org/docs/history.php.

[Perry and Wolf, 1989] Perry, D. E. and Wolf, A. L. (1989). Software architecture. Republished as [Perry and Wolf, 1992]. Online version at http://www.ece.utexas.edu/∼perry/work/papers/swa-sen.pdf.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4).

[Raymond, 2000] Raymond, E. S. (2000). The cathedral and the bazaar. Version 3. Online version at http://www.catb.org/∼esr/writings/cathedral-bazaar/cathedral-bazaar/.

[Rozanski and Woods, 2005] Rozanski, N. and Woods, E. (2005). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional.

[Williams, 1989] Williams, T. J., editor (1989). *A Reference Model For Computer Integrated Manufacturing (CIM): A Description from the Viewpoint of Industrial Automation.* Instrument Society of America.