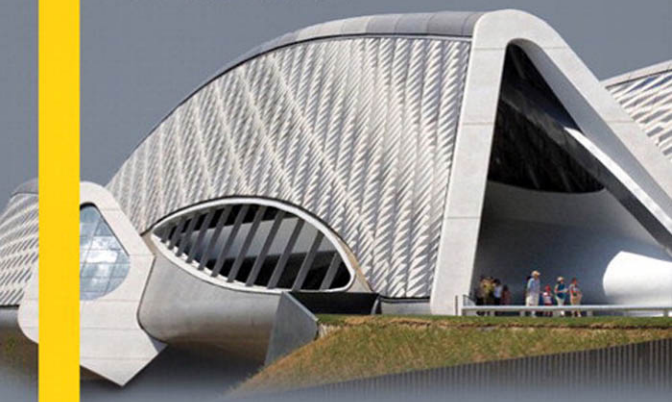
Software Architecture in Practice

Third Edition



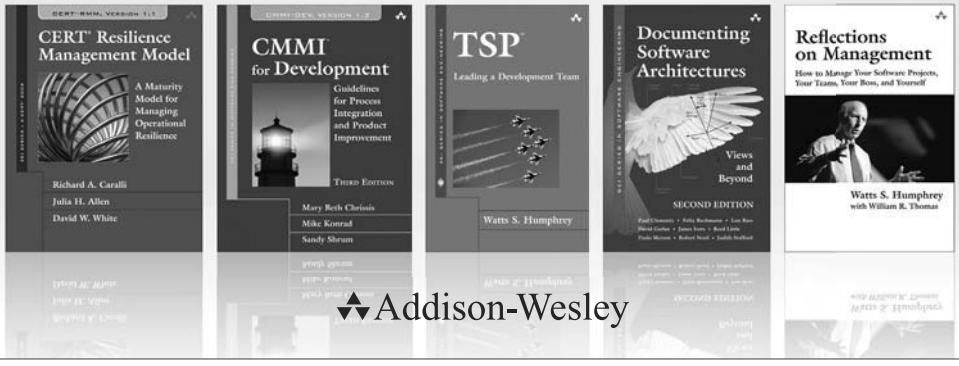
Len Bass · Paul Clements · Rick Kazman

Software Architecture in Practice

Third Edition

The SEI Series in Software Engineering

Software Engineering Institute | Carnegie Mellon



Addison-Wesley

Visit **informit.com/sei** for a complete list of available products.

The **SEI Series in Software Engineering** represents is a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Books in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all books in the series address critical problems in software engineering for which practical solutions are available.

Software Architecture in Practice

Third Edition

Len Bass
Paul Clements
Rick Kazman

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Library of Congress Cataloging-in-Publication Data

Bass, Len.
Software architecture in practice / Len Bass, Paul Clements, Rick Kazman.—3rd ed.
p. cm.—(SEI series in software engineering)
Includes bibliographical references and index.
ISBN 978-0-321-81573-6 (hardcover : alk. paper) 1. Software architecture. 2. System design. I. Clements, Paul, 1955– II. Kazman, Rick. III. Title.
QA76.754.B37 2012
005.1—dc23

2012023744

ISBN-13: 978-0-321-81573-6
ISBN-10: 0-321-81573-4

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, September 2012

Contents

Preface *xv*

Reader's Guide *xvii*

Acknowledgments *xix*

PART ONE INTRODUCTION 1

CHAPTER 1 What Is Software Architecture? 3

1.1 What Software Architecture Is and What It Isn't 4

1.2 Architectural Structures and Views 9

1.3 Architectural Patterns 18

1.4 What Makes a "Good" Architecture? 19

1.5 Summary 21

1.6 For Further Reading 22

1.7 Discussion Questions 23

CHAPTER 2 Why Is Software Architecture Important? 25

2.1 Inhibiting or Enabling a System's Quality Attributes 26

2.2 Reasoning About and Managing Change 27

2.3 Predicting System Qualities 28

2.4 Enhancing Communication among Stakeholders 29

2.5 Carrying Early Design Decisions 31

2.6 Defining Constraints on an Implementation 32

2.7 Influencing the Organizational Structure 33

2.8 Enabling Evolutionary Prototyping 33

2.9 Improving Cost and Schedule Estimates 34
2.10 Supplying a Transferable, Reusable Model 35
2.11 Allowing Incorporation of Independently Developed Components 35
2.12 Restricting the Vocabulary of Design Alternatives 36
2.13 Providing a Basis for Training 37
2.14 Summary 37
2.15 For Further Reading 38
2.16 Discussion Questions 38

CHAPTER 3 The Many Contexts of Software Architecture 39
3.1 Architecture in a Technical Context 40
3.2 Architecture in a Project Life-Cycle Context 44
3.3 Architecture in a Business Context 49
3.4 Architecture in a Professional Context 51
3.5 Stakeholders 52
3.6 How Is Architecture Influenced? 56
3.7 What Do Architectures Influence? 57
3.8 Summary 59
3.9 For Further Reading 59
3.10 Discussion Questions 60

PART TWO QUALITY ATTRIBUTES 61

CHAPTER 4 Understanding Quality Attributes 63
4.1 Architecture and Requirements 64
4.2 Functionality 65
4.3 Quality Attribute Considerations 65
4.4 Specifying Quality Attribute Requirements 68
4.5 Achieving Quality Attributes through Tactics 70
4.6 Guiding Quality Design Decisions 72
4.7 Summary 76

4.8 For Further Reading 77
4.9 Discussion Questions 77

CHAPTER 5 Availability 79
5.1 Availability General Scenario 85
5.2 Tactics for Availability 87
5.3 A Design Checklist for Availability 96
5.4 Summary 98
5.5 For Further Reading 99
5.6 Discussion Questions 100

CHAPTER 6 Interoperability 103
6.1 Interoperability General Scenario 107
6.2 Tactics for Interoperability 110
6.3 A Design Checklist for Interoperability 114
6.4 Summary 115
6.5 For Further Reading 116
6.6 Discussion Questions 116

CHAPTER 7 Modifiability 117
7.1 Modifiability General Scenario 119
7.2 Tactics for Modifiability 121
7.3 A Design Checklist for Modifiability 125
7.4 Summary 128
7.5 For Further Reading 128
7.6 Discussion Questions 128

CHAPTER 8 Performance 131
8.1 Performance General Scenario 132
8.2 Tactics for Performance 135
8.3 A Design Checklist for Performance 142
8.4 Summary 145
8.5 For Further Reading 145
8.6 Discussion Questions 145

CHAPTER 9 Security 147
9.1 Security General Scenario 148
9.2 Tactics for Security 150

9.3 A Design Checklist for Security 154
9.4 Summary 156
9.5 For Further Reading 157
9.6 Discussion Questions 158

CHAPTER 10 Testability 159
10.1 Testability General Scenario 162
10.2 Tactics for Testability 164
10.3 A Design Checklist for Testability 169
10.4 Summary 172
10.5 For Further Reading 172
10.6 Discussion Questions 173

CHAPTER 11 Usability 175
11.1 Usability General Scenario 176
11.2 Tactics for Usability 177
11.3 A Design Checklist for Usability 181
11.4 Summary 183
11.5 For Further Reading 183
11.6 Discussion Questions 183

CHAPTER 12 Other Quality Attributes 185
12.1 Other Important Quality Attributes 185
12.2 Other Categories of Quality Attributes 189
12.3 Software Quality Attributes and System Quality Attributes 190
12.4 Using Standard Lists of Quality Attributes—or Not 193
12.5 Dealing with “X-ability”: Bringing a New Quality Attribute into the Fold 196
12.6 For Further Reading 200
12.7 Discussion Questions 201

CHAPTER 13 Architectural Tactics and Patterns 203
13.1 Architectural Patterns 204
13.2 Overview of the Patterns Catalog 205
13.3 Relationships between Tactics and Patterns 238

13.4 Using Tactics Together 242
13.5 Summary 247
13.6 For Further Reading 248
13.7 Discussion Questions 249

CHAPTER 14 Quality Attribute Modeling and Analysis 251
14.1 Modeling Architectures to Enable Quality Attribute Analysis 252
14.2 Quality Attribute Checklists 260
14.3 Thought Experiments and Back-of-the-Envelope Analysis 262
14.4 Experiments, Simulations, and Prototypes 264
14.5 Analysis at Different Stages of the Life Cycle 265
14.6 Summary 266
14.7 For Further Reading 267
14.8 Discussion Questions 269

PART THREE ARCHITECTURE IN THE LIFE CYCLE 271

CHAPTER 15 Architecture in Agile Projects 275
15.1 How Much Architecture? 277
15.2 Agility and Architecture Methods 281
15.3 A Brief Example of Agile Architecting 283
15.4 Guidelines for the Agile Architect 286
15.5 Summary 287
15.6 For Further Reading 288
15.7 Discussion Questions 289

CHAPTER 16 Architecture and Requirements 291
16.1 Gathering ASRs from Requirements Documents 292
16.2 Gathering ASRs by Interviewing Stakeholders 294
16.3 Gathering ASRs by Understanding the Business Goals 296

16.4 Capturing ASRs in a Utility Tree 304
16.5 Tying the Methods Together 308
16.6 Summary 308
16.7 For Further Reading 309
16.8 Discussion Questions 309

CHAPTER 17 Designing an Architecture 311
17.1 Design Strategy 311
17.2 The Attribute-Driven Design Method 316
17.3 The Steps of ADD 318
17.4 Summary 325
17.5 For Further Reading 325
17.6 Discussion Questions 326

CHAPTER 18 Documenting Software Architectures 327
18.1 Uses and Audiences for Architecture Documentation 328
18.2 Notations for Architecture Documentation 329
18.3 Views 331
18.4 Choosing the Views 341
18.5 Combining Views 343
18.6 Building the Documentation Package 345
18.7 Documenting Behavior 351
18.8 Architecture Documentation and Quality Attributes 354
18.9 Documenting Architectures That Change Faster Than You Can Document Them 355
18.10 Documenting Architecture in an Agile Development Project 356
18.11 Summary 359
18.12 For Further Reading 360
18.13 Discussion Questions 360

CHAPTER 19 Architecture, Implementation, and Testing 363
19.1 Architecture and Implementation 363
19.2 Architecture and Testing 370

19.3 Summary 376
19.4 For Further Reading 376
19.5 Discussion Questions 377

CHAPTER 20 Architecture Reconstruction and Conformance 379

20.1 Architecture Reconstruction Process 381
20.2 Raw View Extraction 382
20.3 Database Construction 386
20.4 View Fusion 388
20.5 Architecture Analysis: Finding Violations 389
20.6 Guidelines 392
20.7 Summary 393
20.8 For Further Reading 394
20.9 Discussion Questions 395

CHAPTER 21 Architecture Evaluation 397

21.1 Evaluation Factors 397
21.2 The Architecture Tradeoff Analysis Method 400
21.3 Lightweight Architecture Evaluation 415
21.4 Summary 417
21.5 For Further Reading 417
21.6 Discussion Questions 418

CHAPTER 22 Management and Governance 419

22.1 Planning 420
22.2 Organizing 422
22.3 Implementing 427
22.4 Measuring 429
22.5 Governance 430
22.6 Summary 432
22.7 For Further Reading 432
22.8 Discussion Questions 433

PART FOUR ARCHITECTURE AND BUSINESS 435

CHAPTER 23 Economic Analysis of Architectures 437

23.1 Decision-Making Context 438
23.2 The Basis for the Economic Analyses 439
23.3 Putting Theory into Practice: The CBAM 442
23.4 Case Study: The NASA ECS Project 450
23.5 Summary 457
23.6 For Further Reading 458
23.7 Discussion Questions 458

CHAPTER 24 Architecture Competence 459

24.1 Competence of Individuals: Duties, Skills, and Knowledge of Architects 460
24.2 Competence of a Software Architecture Organization 467
24.3 Summary 475
24.4 For Further Reading 475
24.5 Discussion Questions 477

CHAPTER 25 Architecture and Software Product Lines 479

25.1 An Example of Product Line Variability 482
25.2 What Makes a Software Product Line Work? 483
25.3 Product Line Scope 486
25.4 The Quality Attribute of Variability 488
25.5 The Role of a Product Line Architecture 488
25.6 Variation Mechanisms 490
25.7 Evaluating a Product Line Architecture 493
25.8 Key Software Product Line Issues 494
25.9 Summary 497
25.10 For Further Reading 498
25.11 Discussion Questions 498

PART FIVE THE BRAVE NEW WORLD 501

CHAPTER 26 Architecture in the Cloud 503

26.1 Basic Cloud Definitions 504
26.2 Service Models and Deployment Options 505
26.3 Economic Justification 506
26.4 Base Mechanisms 509
26.5 Sample Technologies 514
26.6 Architecting in a Cloud Environment 520
26.7 Summary 524
26.8 For Further Reading 524
26.9 Discussion Questions 525

CHAPTER 27 Architectures for the Edge 527

27.1 The Ecosystem of Edge-Dominant Systems 528
27.2 Changes to the Software Development Life Cycle 530
27.3 Implications for Architecture 531
27.4 Implications of the Metropolis Model 533
27.5 Summary 537
27.6 For Further Reading 538
27.7 Discussion Questions 538

CHAPTER 28 Epilogue 541

References 547

About the Authors 561

Index 563

This page intentionally left blank

Preface

I should have no objection to go over the same life from its beginning to the end: requesting only the advantage authors have, of correcting in a [third] edition the faults of the first [two].
— Benjamin Franklin

It has been a decade since the publication of the second edition of this book. During that time, the field of software architecture has broadened its focus from being primarily internally oriented—How does one design, evaluate, and document software?—to including external impacts as well—a deeper understanding of the influences on architectures and a deeper understanding of the impact architectures have on the life cycle, organizations, and management.

The past ten years have also seen dramatic changes in the types of systems being constructed. Large data, social media, and the cloud are all areas that, at most, were embryonic ten years ago and now are not only mature but extremely influential.

We listened to some of the criticisms of the previous editions and have included much more material on patterns, reorganized the material on quality attributes, and made interoperability a quality attribute worthy of its own chapter. We also provide guidance about how you can generate scenarios and tactics for your own favorite quality attributes.

To accommodate this plethora of new material, we had to make difficult choices. In particular, this edition of the book does not include extended case studies as the prior editions did. This decision also reflects the maturing of the field, in the sense that case studies about the choices made in software architectures are more prevalent than they were ten years ago, and they are less necessary to convince readers of the importance of software architecture. The case studies from the first two editions are available, however, on the book's website, at www.informit.com/title/9780321815736. In addition, on the same website, we have slides that will assist instructors in presenting this material.

We have thoroughly reworked many of the topics covered in this edition. In particular, we realize that the methods we present—for architecture design, analysis, and documentation—are one version of how to achieve a particular goal, but there are others. This led us to separate the methods that we present

in detail from their underlying theory. We now present the theory first with specific methods given as illustrations of possible realizations of the theories. The new topics in this edition include architecture-centric project management; architecture competence; requirements modeling and analysis; Agile methods; implementation and testing; the cloud; and the edge.

As with the prior editions, we firmly believe that the topics are best discussed in either reading groups or in classroom settings, and to that end we have included a collection of discussion questions at the end of each chapter. Most of these questions are open-ended, with no absolute right or wrong answers, so you, as a reader, should emphasize how you justify your answer rather than just answer the question itself.

Reader's Guide

We have structured this book into five distinct portions. Part One introduces architecture and the various contextual lenses through which it could be viewed. These are the following:

- *Technical.* What technical role does the software architecture play in the system or systems of which it's a part?
- *Project.* How does a software architecture relate to the other phases of a software development life cycle?
- *Business.* How does the presence of a software architecture affect an organization's business environment?
- *Professional.* What is the role of a software architect in an organization or a development project?

Part Two is focused on technical background. Part Two describes how decisions are made. Decisions are based on the desired quality attributes for a system, and Chapters 5–11 describe seven different quality attributes and the techniques used to achieve them. The seven are availability, interoperability, maintainability, performance, security, testability, and usability. Chapter 12 tells you how to add other quality attributes to our seven, Chapter 13 discusses patterns and tactics, and Chapter 14 discusses the various types of modeling and analysis that are possible.

Part Three is devoted to how a software architecture is related to the other portions of the life cycle. Of special note is how architecture can be used in Agile projects. We discuss individually other aspects of the life cycle: requirements, design, implementation and testing, recovery and conformance, and evaluation.

Part Four deals with the business of architecting from an economic perspective, from an organizational perspective, and from the perspective of constructing a series of similar systems.

Part Five discusses several important emerging technologies and how architecture relates to these technologies.

This page intentionally left blank

Acknowledgments

We had a fantastic collection of reviewers for this edition, and their assistance helped make this a better book. Our reviewers were Muhammad Ali Babar, Felix Bachmann, Joe Batman, Phil Bianco, Jeromy Carriere, Roger Champagne, Steve Chenoweth, Viktor Clerc, Andres Diaz Pace, George Fairbanks, Rik Farenhorst, Ian Gorton, Greg Hartman, Rich Hilliard, James Ivers, John Klein, Philippe Kruchten, Phil Laplante, George Leih, Grace Lewis, John McGregor, Tommi Mikkonen, Linda Northrop, Ipek Ozkaya, Eltjo Poort, Eelco Rommes, Nick Rozanski, Jungwoo Ryoo, James Scott, Antony Tang, Arjen Uittenbogaard, Hans van Vliet, Hiroshi Wada, Rob Wojcik, Eoin Woods, and Liming Zhu.

In addition, we had significant contributions from Liming Zhu, Hong-Mei Chen, Jungwoo Ryoo, Phil Laplante, James Scott, Grace Lewis, and Nick Rozanski that helped give the book a richer flavor than one written by just the three of us.

The issue of build efficiency in Chapter 12 came from Rolf Siegers and John McDonald of Raytheon. John Klein and Eltjo Poort contributed the "abstract system clock" and "sandbox mode" tactics, respectively, for testability. The list of stakeholders in Chapter 3 is from *Documenting Software Architectures: Views and Beyond, Second Edition.* Some of the material in Chapter 28 was inspired by a talk given by Anthony Lattanze called "Organizational Design Thinking" in 2011.

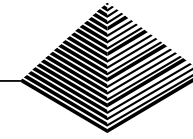
Joe Batman was instrumental in the creation of the seven categories of design decisions we describe in Chapter 4. In addition, the descriptions of the security view, communications view, and exception view in Chapter 18 are based on material that Joe wrote while planning the documentation for a real system's architecture. Much of the new material on modifiability tactics was based on the work of Felix Bachmann and Rod Nord. James Ivers helped us with the security tactics.

Both Paul Clements and Len Bass have taken new positions since the last edition was published, and we thank their new respective managements (BigLever Software for Paul and NICTA for Len) for their willingness to support our work on this edition. We would also like to thank our (former) colleagues at the Software Engineering Institute for multiple contributions to the evolution of the ideas expressed in this edition.

Finally, as always, we thank our editor at Addison-Wesley, Peter Gordon, for providing guidance and support during the writing and production processes.

This page intentionally left blank

4



Understanding Quality Attributes

Between stimulus and response, there is a space. In that space is our power to choose our response. In our response lies our growth and our freedom.
— Viktor E. Frankl, *Man's Search for Meaning*

As we have seen in the Architecture Influence Cycle (in Chapter 3), many factors determine the qualities that must be provided for in a system's architecture. These qualities go beyond functionality, which is the basic statement of the system's capabilities, services, and behavior. Although functionality and other qualities are closely related, as you will see, functionality often takes the front seat in the development scheme. This preference is shortsighted, however. Systems are frequently redesigned not because they are functionally deficient—the replacements are often functionally identical—but because they are difficult to maintain, port, or scale; or they are too slow; or they have been compromised by hackers. In Chapter 2, we said that architecture was the first place in software creation in which quality requirements could be addressed. It is the mapping of a system's functionality onto software structures that determines the architecture's support for qualities. In Chapters 5–11 we discuss how various qualities are supported by architectural design decisions. In Chapter 17 we show how to integrate all of the quality attribute decisions into a single design.

We have been using the term "quality attribute" loosely, but now it is time to define it more carefully. A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders. You can think of a quality attribute as measuring the "goodness" of a product along some dimension of interest to a stakeholder.

In this chapter our focus is on understanding the following:

- How to express the qualities we want our architecture to provide to the system or systems we are building from it

- How to achieve those qualities
- How to determine the design decisions we might make with respect to those qualities

This chapter provides the context for the discussion of specific quality attributes in Chapters 5–11.

4.1 Architecture and Requirements

Requirements for a system come in a variety of forms: textual requirements, mockups, existing systems, use cases, user stories, and more. Chapter 16 discusses the concept of an *architecturally significant requirement*, the role such requirements play in architecture, and how to identify them. No matter the source, all requirements encompass the following categories:

1. *Functional requirements.* These requirements state what the system must do, and how it must behave or react to runtime stimuli.
2. *Quality attribute requirements.* These requirements are qualifications of the functional requirements or of the overall product. A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input. A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.
3. *Constraints.* A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that's already been made. Examples include the requirement to use a certain programming language or to reuse a certain existing module, or a management fiat to make your system service oriented. These choices are arguably in the purview of the architect, but external factors (such as not being able to train the staff in a new language, or having a business agreement with a software supplier, or pushing business goals of service interoperability) have led those in power to dictate these design outcomes.

What is the "response" of architecture to each of these kinds of requirements?

1. Functional requirements are satisfied by assigning an appropriate sequence of responsibilities throughout the design. As we will see later in this chapter, assigning responsibilities to architectural elements is a fundamental architectural design decision.
2. Quality attribute requirements are satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures. Chapter 17 will show this approach in more detail.

3. Constraints are satisfied by accepting the design decision and reconciling it with other affected design decisions.

4.2 Functionality

Functionality is the ability of the system to do the work for which it was intended. Of all of the requirements, functionality has the strangest relationship to architecture.

First of all, functionality does not determine architecture. That is, given a set of required functionality, there is no end to the architectures you could create to satisfy that functionality. At the very least, you could divide up the functionality in any number of ways and assign the subpieces to different architectural elements.

In fact, if functionality were the only thing that mattered, you wouldn't have to divide the system into pieces at all; a single monolithic blob with no internal structure would do just fine. Instead, we design our systems as structured sets of cooperating architectural elements—modules, layers, classes, services, databases, apps, threads, peers, tiers, and on and on—to make them understandable and to support a variety of other purposes. Those "other purposes" are the other quality attributes that we'll turn our attention to in the remaining sections of this chapter, and the remaining chapters of Part II.

But although functionality is independent of any particular structure, functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures.

Although responsibilities can be allocated arbitrarily to any modules, software architecture constrains this allocation when other quality attributes are important. For example, systems are frequently divided so that several people can cooperatively build them. The architect's interest in functionality is in how it interacts with and constrains other qualities.

4.3 Quality Attribute Considerations

Just as a system's functions do not stand on their own without due consideration of other quality attributes, neither do quality attributes stand on their own; they pertain to the functions of the system. If a functional requirement is "When the user presses the green button, the Options dialog appears," a performance QA annotation might describe how quickly the dialog will appear; an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired; a usability QA annotation might describe how easy it is to learn this function.

Functional Requirements

After more than 15 years of writing and discussing the distinction between functional requirements and quality requirements, the definition of functional requirements still eludes me. Quality attribute requirements are well defined: performance has to do with the timing behavior of the system, modifiability has to do with the ability of the system to support changes in its behavior or other qualities after initial deployment, availability has to do with the ability of the system to survive failures, and so forth.

Function, however, is much more slippery. An international standard (ISO 25010) defines functional suitability as "the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions." That is, functionality is the ability to provide functions. One interpretation of this definition is that functionality describes what the system does and quality describes how well the system does its function. That is, qualities are attributes of the system and function is the purpose of the system.

This distinction breaks down, however, when you consider the nature of some of the "function." If the function of the software is to control engine behavior, how can the function be correctly implemented without considering timing behavior? Is the ability to control access through requiring a user name/password combination not a function even though it is not the purpose of any system?

I like much better the use of the word "responsibility" to describe computations that a system must perform. Questions such as "What are the timing constraints on that set of responsibilities?", "What modifications are anticipated with respect to that set of responsibilities?", and "What class of users is allowed to execute that set of responsibilities?" make sense and are actionable.

The achievement of qualities induces responsibility; think of the user name/password example just mentioned. Further, one can identify responsibilities as being associated with a particular set of requirements.

So does this mean that the term "functional requirement" shouldn't be used? People have an understanding of the term, but when precision is desired, we should talk about sets of specific responsibilities instead.

Paul Clements has long ranted against the careless use of the term "nonfunctional," and now it's my turn to rant against the careless use of the term "functional"—probably equally ineffectually.

—LB

Quality attributes have been of interest to the software community at least since the 1970s. There are a variety of published taxonomies and definitions, and many of them have their own research and practitioner communities. From an

architect's perspective, there are three problems with previous discussions of system quality attributes:

1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be "modifiable." Every system may be modifiable with respect to one set of changes and not modifiable with respect to another. The other quality attributes are similar in this regard: a system may be robust with respect to some faults and brittle with respect to others. And so forth.
2. Discussion often focuses on which quality a particular concern belongs to. Is a system failure due to a denial-of-service attack an aspect of availability, an aspect of performance, an aspect of security, or an aspect of usability? All four attribute communities would claim ownership of a system failure due to a denial-of-service attack. All are, to some extent, correct. But this doesn't help us, as architects, understand and create architectural solutions to manage the attributes of concern.
3. Each attribute community has developed its own vocabulary. The performance community has "events" arriving at a system, the security community has "attacks" arriving at a system, the availability community has "failures" of a system, and the usability community has "user input." All of these may actually refer to the same occurrence, but they are described using different terms.

A solution to the first two of these problems (untestable definitions and overlapping concerns) is to use *quality attribute scenarios* as a means of characterizing quality attributes (see the next section). A solution to the third problem is to provide a discussion of each attribute—concentrating on its underlying concerns—to illustrate the concepts that are fundamental to that attribute community.

There are two categories of quality attributes on which we focus. The first is those that describe some property of the system at runtime, such as availability, performance, or usability. The second is those that describe some property of the development of the system, such as modifiability or testability.

Within complex systems, quality attributes can never be achieved in isolation. The achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others. For example, almost every quality attribute negatively affects performance. Take portability. The main technique for achieving portable software is to isolate system dependencies, which introduces overhead into the system's execution, typically as process or procedure boundaries, and this hurts performance. Determining the design that satisfies all of the quality attribute requirements is partially a matter of making the appropriate tradeoffs; we discuss design in Chapter 17. Our purpose here is to provide the context for discussing each quality attribute. In particular, we focus on how quality attributes can be specified, what architectural decisions will enable the achievement of particular quality attributes, and what questions about quality attributes will enable the architect to make the correct design decisions.

4.4 Specifying Quality Attribute Requirements

A quality attribute requirement should be unambiguous and testable. We use a common form to specify all quality attribute requirements. This has the advantage of emphasizing the commonalities among all quality attributes. It has the disadvantage of occasionally being a force-fit for some aspects of quality attributes.

Our common form for quality attribute expression has these parts:

- *Stimulus.* We use the term "stimulus" to describe an event arriving at the system. The stimulus can be an event to the performance community, a user operation to the usability community, or an attack to the security community. We use the same term to describe a motivating action for developmental qualities. Thus, a stimulus for modifiability is a request for a modification; a stimulus for testability is the completion of a phase of development.
- *Stimulus source.* A stimulus must have a source—it must come from somewhere. The source of the stimulus may affect how it is treated by the system. A request from a trusted user will not undergo the same scrutiny as a request by an untrusted user.
- *Response.* How the system should respond to the stimulus must also be specified. The response consists of the responsibilities that the system (for runtime qualities) or the developers (for development-time qualities) should perform in response to the stimulus. For example, in a performance scenario, an event arrives (the stimulus) and the system should process that event and generate a response. In a modifiability scenario, a request for a modification arrives (the stimulus) and the developers should implement the modification—without side effects—and then test and deploy the modification.
- *Response measure.* Determining whether a response is satisfactory—whether the requirement is satisfied—is enabled by providing a response measure. For performance this could be a measure of latency or throughput; for modifiability it could be the labor or wall clock time required to make, test, and deploy the modification.

These four characteristics of a scenario are the heart of our quality attribute specifications. But there are two more characteristics that are important: environment and artifact.

- *Environment.* The environment of a requirement is the set of circumstances in which the scenario takes place. The environment acts as a qualifier on the stimulus. For example, a request for a modification that arrives after the code has been frozen for a release may be treated differently than one that arrives before the freeze. A failure that is the fifth successive failure

of a component may be treated differently than the first failure of that component.

- *Artifact.* Finally, the artifact is the portion of the system to which the requirement applies. Frequently this is the entire system, but occasionally specific portions of the system may be called out. A failure in a data store may be treated differently than a failure in the metadata store. Modifications to the user interface may have faster response times than modifications to the middleware.

To summarize how we specify quality attribute requirements, we capture them formally as six-part scenarios. While it is common to omit one or more of these six parts, particularly in the early stages of thinking about quality attributes, knowing that all parts are there forces the architect to consider whether each part is relevant.

In summary, here are the six parts:

1. *Source of stimulus.* This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. *Stimulus.* The stimulus is a condition that requires a response when it arrives at a system.
3. *Environment.* The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, "normal" operation can refer to one of a number of modes. For these kinds of systems, the environment should specify in which mode the system is executing.
4. *Artifact.* Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. *Response.* The response is the activity undertaken as the result of the arrival of the stimulus.
6. *Response measure.* When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

We distinguish *general* quality attribute scenarios (which we call "general scenarios" for short)—those that are system independent and can, potentially, pertain to any system—from *concrete* quality attribute scenarios (concrete scenarios)—those that are specific to the particular system under consideration.

We can characterize quality attributes as a collection of general scenarios. Of course, to translate these generic attribute characterizations into requirements for a particular system, the general scenarios need to be made system specific. Detailed examples of these scenarios will be given in Chapters 5–11. Figure 4.1 shows the parts of a quality attribute scenario that we have just discussed. Figure 4.2 shows an example of a general scenario, in this case for availability.

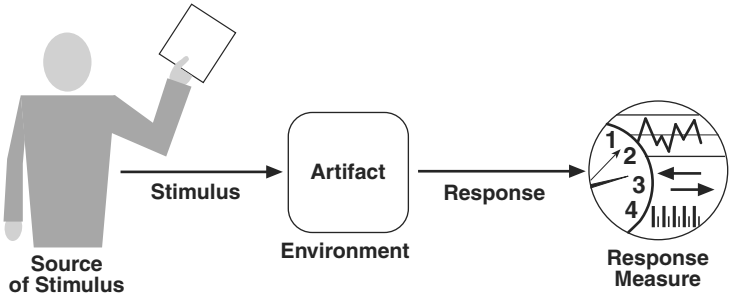


FIGURE 4.1 The parts of a quality attribute scenario

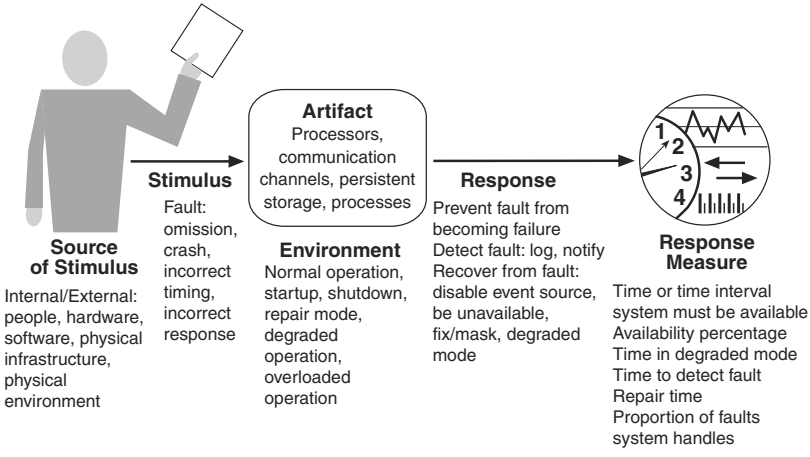


FIGURE 4.2 A general scenario for availability

4.5 Achieving Quality Attributes through Tactics

The quality attribute requirements specify the responses of the system that, with a bit of luck and a dose of good planning, realize the goals of the business. We now turn to the techniques an architect can use to *achieve* the required quality attributes. We call these techniques *architectural tactics.* A tactic is a design decision that influences the achievement of a quality attribute response—tactics directly affect the system's response to some stimulus. Tactics impart portability to one design, high performance to another, and integrability to a third.

Not My Problem

One time I was doing an architecture analysis on a complex system created by and for Lawrence Livermore National Laboratory. If you visit their website (www.llnl.gov) and try to figure out what Livermore Labs does, you will see the word "security" mentioned over and over. The lab focuses on nuclear security, international and domestic security, and environmental and energy security. Serious stuff . . .

Keeping this emphasis in mind, I asked them to describe the quality attributes of concern for the system that I was analyzing. I'm sure you can imagine my surprise when security wasn't mentioned once! The system stakeholders mentioned performance, modifiability, evolvability, interoperability, configurability, and portability, and one or two more, but the word security never passed their lips.

Being a good analyst, I questioned this seemingly shocking and obvious omission. Their answer was simple and, in retrospect, straightforward: "We don't care about it. Our systems are not connected to any external network and we have barbed-wire fences and guards with machine guns." Of course, *someone* at Livermore Labs was very interested in security. But it was clearly not the software architects.

—RK

The focus of a tactic is on a single quality attribute response. Within a tactic, there is no consideration of tradeoffs. Tradeoffs must be explicitly considered and controlled by the designer. In this respect, tactics differ from architectural patterns, where tradeoffs are built into the pattern. (We visit the relation between tactics and patterns in Chapter 14. Chapter 13 explains how sets of tactics for a quality attribute can be constructed, which are the steps we used to produce the set in this book.)

A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality. We represent the relationship between stimulus, tactics, and response in Figure 4.3. The tactics, like design patterns, are design techniques that architects have been using for years. Our contribution is to isolate, catalog, and describe them. We are not inventing tactics here, we are just capturing what architects do in practice.

Why do we do this? There are three reasons:

1. Design patterns are complex; they typically consist of a bundle of design decisions. But patterns are often difficult to apply as is; architects need to modify and adapt them. By understanding the role of tactics, an architect can more easily assess the options for augmenting an existing pattern to achieve a quality attribute goal.

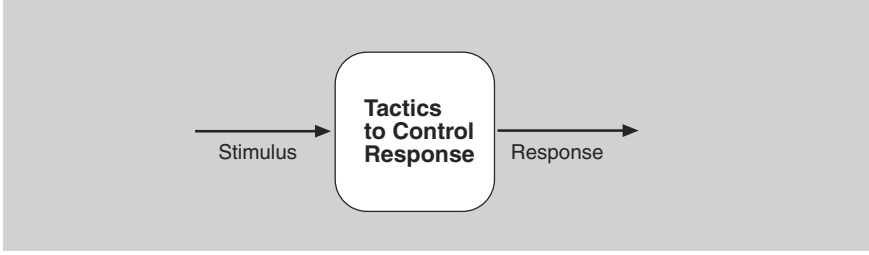


FIGURE 4.3 Tactics are intended to control responses to stimuli.

2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles." Tactics give the architect insight into the properties of the resulting design fragment.
3. By cataloging tactics, we provide a way of making design more systematic within some limitations. Our list of tactics does not provide a taxonomy. We only provide a categorization. The tactics will overlap, and you frequently will have a choice among multiple tactics to improve a particular quality attribute. The choice of which tactic to use depends on factors such as tradeoffs among other quality attributes and the cost to implement. These considerations transcend the discussion of tactics for particular quality attributes. Chapter 17 provides some techniques for choosing among competing tactics.

The tactics that we present can and should be refined. Consider performance: *Schedule resources* is a common performance tactic. But this tactic needs to be refined into a specific scheduling strategy, such as shortest-job-first, round-robin, and so forth, for specific purposes. *Use an intermediary* is a modifiability tactic. But there are multiple types of intermediaries (layers, brokers, and proxies, to name just a few). Thus there are refinements that a designer will employ to make each tactic concrete.

In addition, the application of a tactic depends on the context. Again considering performance: *Manage sampling rate* is relevant in some real-time systems but not in all real-time systems and certainly not in database systems.

4.6 Guiding Quality Design Decisions

Recall that one can view an architecture as the result of applying a collection of design decisions. What we present here is a systematic categorization of these

decisions so that an architect can focus attention on those design dimensions likely to be most troublesome.

The seven categories of design decisions are

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

These categories are not the only way to classify architectural design decisions, but they do provide a rational division of concerns. These categories might overlap, but it's all right if a particular decision exists in two different categories, because the concern of the architect is to ensure that every important decision is considered. Our categorization of decisions is partially based on our definition of software architecture in that many of our categories relate to the definition of structures and the relations among them.

Allocation of Responsibilities

Decisions involving allocation of responsibilities include the following:

- Identifying the important responsibilities, including basic system functions, architectural infrastructure, and satisfaction of quality attributes.
- Determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).

Strategies for making these decisions include functional decomposition, modeling real-world objects, grouping based on the major modes of system operation, or grouping based on similar quality requirements: processing frame rate, security level, or expected changes.

In Chapters 5–11, where we apply these design decision categories to a number of important quality attributes, the checklists we provide for the allocation of responsibilities category is derived systematically from understanding the stimuli and responses listed in the general scenario for that QA.

Coordination Model

Software works by having elements interact with each other through designed mechanisms. These mechanisms are collectively referred to as a coordination model. Decisions about the coordination model include these:

- Identifying the elements of the system that must coordinate, or are prohibited from coordinating.
- Determining the properties of the coordination, such as timeliness, currency, completeness, correctness, and consistency.
- Choosing the communication mechanisms (between systems, between our system and external entities, between elements of our system) that realize those properties. Important properties of the communication mechanisms include stateful versus stateless, synchronous versus asynchronous, guaranteed versus nonguaranteed delivery, and performance-related properties such as throughput and latency.

Data Model

Every system must represent artifacts of system-wide interest—data—in some internal fashion. The collection of those representations and how to interpret them is referred to as the data model. Decisions about the data model include the following:

- Choosing the major data abstractions, their operations, and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated, and destroyed.
- Compiling metadata needed for consistent interpretation of the data.
- Organizing the data. This includes determining whether the data is going to be kept in a relational database, a collection of objects, or both. If both, then the mapping between the two different locations of the data must be determined.

Management of Resources

An architect may need to arbitrate the use of shared resources in the architecture. These include hard resources (e.g., CPU, memory, battery, hardware buffers, system clock, I/O ports) and soft resources (e.g., system locks, software buffers, thread pools, and non-thread-safe code).

Decisions for management of resources include the following:

- Identifying the resources that must be managed and determining the limits for each.
- Determining which system element(s) manage each resource.
- Determining how resources are shared and the arbitration strategies employed when there is contention.
- Determining the impact of saturation on different resources. For example, as a CPU becomes more heavily loaded, performance usually just degrades fairly steadily. On the other hand, when you start to run out of memory, at

some point you start paging/swapping intensively and your performance suddenly crashes to a halt.

Mapping among Architectural Elements

An architecture must provide two types of mappings. First, there is mapping between elements in different types of architecture structures—for example, mapping from units of development (modules) to units of execution (threads or processes). Next, there is mapping between software elements and environment elements—for example, mapping from processes to the specific CPUs where these processes will execute.

Useful mappings include these:

- The mapping of modules and runtime elements to each other—that is, the runtime elements that are created from each module; the modules that contain the code for each runtime element.
- The assignment of runtime elements to processors.
- The assignment of items in the data model to data stores.
- The mapping of modules and runtime elements to units of delivery.

Binding Time Decisions

Binding time decisions introduce allowable ranges of variation. This variation can be bound at different times in the software life cycle by different entities—from design time by a developer to runtime by an end user. A binding time decision establishes the scope, the point in the life cycle, and the mechanism for achieving the variation.

The decisions in the other six categories have an associated binding time decision. Examples of such binding time decisions include the following:

- For allocation of responsibilities, you can have build-time selection of modules via a parameterized makefile.
- For choice of coordination model, you can design runtime negotiation of protocols.
- For resource management, you can design a system to accept new peripheral devices plugged in at runtime, after which the system recognizes them and downloads and installs the right drivers automatically.
- For choice of technology, you can build an app store for a smartphone that automatically downloads the version of the app appropriate for the phone of the customer buying the app.

When making binding time decisions, you should consider the costs to implement the decision and the costs to make a modification after you have implemented the decision. For example, if you are considering changing platforms

at some time after code time, you can insulate yourself from the effects caused by porting your system to another platform at some cost. Making this decision depends on the costs incurred by having to modify an early binding compared to the costs incurred by implementing the mechanisms involved in the late binding.

Choice of Technology

Every architecture decision must eventually be realized using a specific technology. Sometimes the technology selection is made by others, before the intentional architecture design process begins. In this case, the chosen technology becomes a constraint on decisions in each of our seven categories. In other cases, the architect must choose a suitable technology to realize a decision in every one of the categories.

Choice of technology decisions involve the following:

- Deciding which technologies are available to realize the decisions made in the other categories.
- Determining whether the available tools to support this technology choice (IDEs, simulators, testing tools, etc.) are adequate for development to proceed.
- Determining the extent of internal familiarity as well as the degree of external support available for the technology (such as courses, tutorials, examples, and availability of contractors who can provide expertise in a crunch) and deciding whether this is adequate to proceed.
- Determining the side effects of choosing a technology, such as a required coordination model or constrained resource management opportunities.
- Determining whether a new technology is compatible with the existing technology stack. For example, can the new technology run on top of or alongside the existing technology stack? Can it communicate with the existing technology stack? Can the new technology be monitored and managed?

4.7 Summary

Requirements for a system come in three categories:

1. *Functional.* These requirements are satisfied by including an appropriate set of responsibilities within the design.
2. *Quality attribute.* These requirements are satisfied by the structures and behaviors of the architecture.
3. *Constraints.* A constraint is a design decision that's already been made.

To express a quality attribute requirement, we use a quality attribute scenario. The parts of the scenario are these:

1. Source of stimulus
2. Stimulus
3. Environment
4. Artifact
5. Response
6. Response measure

An architectural tactic is a design decision that affects a quality attribute response. The focus of a tactic is on a single quality attribute response. Architectural patterns can be seen as "packages" of tactics.

The seven categories of architectural design decisions are these:

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

4.8 For Further Reading

Philippe Kruchten [Kruchten 04] provides another categorization of design decisions.

Pena [Pena 87] uses categories of Function/Form/Economy/Time as a way of categorizing design decisions.

Binding time and mechanisms to achieve different types of binding times are discussed in [Bachmann 05].

Taxonomies of quality attributes can be found in [Boehm 78], [McCall 77], and [ISO 11].

Arguments for viewing architecture as essentially independent from function can be found in [Shaw 95].

4.9 Discussion Questions

1. What is the relationship between a use case and a quality attribute scenario? If you wanted to add quality attribute information to a use case, how would you do it?

2. Do you suppose that the set of tactics for a quality attribute is finite or infinite? Why?
3. Discuss the choice of programming language (an example of choice of technology) and its relation to architecture in general, and the design decisions in the other six categories? For instance, how can certain programming languages enable or inhibit the choice of particular coordination models?
4. We will be using the automatic teller machine as an example throughout the chapters on quality attributes. Enumerate the set of responsibilities that an automatic teller machine should support and propose an initial design to accommodate that set of responsibilities. Justify your proposal.
5. Think about the screens that your favorite automatic teller machine uses. What do those screens tell you about binding time decisions reflected in the architecture?
6. Consider the choice between synchronous and asynchronous communication (a choice in the coordination mechanism category). What quality attribute requirements might lead you to choose one over the other?
7. Consider the choice between stateful and stateless communication (a choice in the coordination mechanism category). What quality attribute requirements might lead you to choose one over the other?
8. Most peer-to-peer architecture employs late binding of the topology. What quality attributes does this promote or inhibit?

Index

AADL (Architecture Analysis and Design Language), 354
Abstract common services tactic, 124
Abstract data sources for testability, 165
Abstract Syntax Tree (AST) analyzers, 386
Abstraction, architecture as, 5–6
Acceptance testing, 372
Access
  basis sets, 261
  network, 504
access_read relationship, 384
access_write relationship, 384
ACID (atomic, consistent, isolated, and durable) properties, 95
Acknowledged system of systems, 106
Active redundancy, 91, 256–259
ActiveMQ product, 224
Activities
  competence, 468
  test, 374–375
Activity diagrams for traces, 353
Actors tactic, 152–153
Adams, Douglas, 437
ADD method. *See* Attribute-Driven Design (ADD) method
Add-ons, 491–492
ADLs (architecture description languages), 330
Adolphus, Gustavus, 42
Adoption strategies, 494–496
Adventure Builder system, 224, 226, 237
Aggregation for usability, 180
Agile projects, 533
  architecture example, 283–285
  architecture methods, 281–283
  architecture overview, 277–281
  description, 44–45
  documenting, 356–357
  guidelines, 286–287
  introduction, 275–277
  patterns, 238
  requirements, 56
  summary, 287–288
AIC (Architecture Influence Cycle)
  description, 58
  Vasa ship, 43
Air France flight 447, 192
Air traffic control systems, 366–367
Allen, Woody, 79
Allocated to relation
  allocation views, 339–340
  deployment structure, 14
  multi-tier pattern, 237
Allocation of responsibilities category
  ASRs, 293
  availability, 96
  interoperability, 114
  modifiability, 126
  performance, 143
  quality design decisions, 73
  security, 154
  testability, 169
  usability, 181
Allocation patterns
  map-reduce, 232–235
  miscellaneous, 238
  multi-tier, 235–237
Allocation structures, 5, 11, 14
Allocation views, 339–340
Allowed-to-use relationship, 206–207
Alpha testing, 372
Alternatives, evaluating, 398
Amazon service-level agreements, 81, 522
Analysis
  architecture, 47–48
  ATAM, 408–409, 411
  availability, 255–259
  back-of-the-envelope, 262–264
  conformance by, 389–392
  economic. *See* Economic analysis
  outsider, 399
  performance, 252–255
Analysts, 54
Analytic model space, 259–260
Analytic perspective on up-front work vs. agility, 279–281
Analytic redundancy tactic, 90
AND gate symbol, 84
Anonymizing test data, 171
Antimissile system, 104
Apache web server, 528, 531
Approaches
  ATAM, 407–409, 411
  CIA, 147–148
  Lightweight Architecture Evaluation, 416

Architects
background and experience, 51–52
cloud environments, 520–523
communication with, 29
competence, 459–467
description and interests, 54
duties, 460–464
knowledge, 466–467
responsibilities, 422–423
skills, 463, 465
test role, 375–376
Architectural structures
allocation, 14
component-and-connector, 13–14
documentation, 17–18
insight from, 11–12
kinds, 10–11
limiting, 17
module, 12–13
relating to each other, 14, 16–17
selecting, 17
table of, 15
views, 9–10
Architecturally significant requirements (ASRs), 46–47, 291–292
ADD method, 320–321
from business goals, 296–304
designing to, 311–312
interviewing stakeholders, 294–296
from requirements documents, 292–293
utility trees for, 304–307
Architecture
Agile projects. *See* Agile projects
analyzing, 47–48
availability. *See* Availability
business context, 49–51
changes, 27–28
cloud. *See* Cloud environments
competence. *See* Competence
conceptual integrity of, 189
design. *See* Design and design strategy
documenting. *See* Documentation
drivers in PALM, 305
economics. *See* Economic analysis
evaluation. *See* Evaluation
implementation. *See* Implementation
influences, 56–58
in life cycle, 271–274
management. *See* Management and governance
modifiability. *See* Modifiability
patterns. *See* Patterns
performance. *See* Performance
product lines. *See* Software product lines
product reuse, 483–484
QAW drivers, 295
QAW plan presentation, 295
quality attributes. *See* Quality attributes
reconstruction and conformance. *See* Reconstruction and conformance
requirements. *See* Architecturally significant requirements (ASRs); Requirements
security. *See* Security
structures. *See* Architectural structures
tactics. *See* Tactics
testability. *See* Testability
usability. *See* Usability
Architecture Analysis and Design Language (AADL), 354
Architecture-centric projects, 279
Architecture description languages (ADLs), 330
Architecture Influence Cycle (AIC)
description, 58
Vasa ship, 43
Architecture Tradeoff Analysis Method (ATAM), 48, 283, 400
approaches, 407–409, 411
business drivers, 404–405
example exercise, 411–414
outputs, 402–403
participants, 400–401
phases, 403–404
presentation, 403–406
results, 411
scenarios, 408, 410
steps, 404–411
Ariane 5 explosion, 192
Aristotle, 185
Arrival pattern for events, 133
Artifacts
availability, 85–86
in evaluation, 399
interoperability, 107–108
modifiability, 119–120
performance, 134
product reuse, 484
quality attributes expressions, 69–70
security, 148, 150
testability, 162–163
usability, 176
variability, 489
ASP.NET framework, 215
Aspects
for testability, 167

variation mechanism, 492
ASRs. *See* Architecturally significant requirements (ASRs)
Assembly connectors in UML, 369
Assertions for system state, 166
Assessment goals, 469
Assessment of competence, 469–472, 474–475
Assign utility
CBAM, 446
NASA ECS project, 452
AST (Abstract Syntax Tree) analyzers, 386
Asymmetric flow in client-server pattern, 218
Asynchronous messaging, 223, 225
ATAM. *See* Architecture Tradeoff Analysis Method (ATAM)
ATM (automatic teller machine) banking system, 219
Atomic, consistent, isolated, and durable (ACID) properties, 95
Attachment relation
broker pattern, 211
client-server pattern, 218
component-and-connector structures, 13
pipe-and-filter pattern, 216
publish-subscribe pattern, 227
shared-data pattern, 231
Attachments in component-and-connector views, 336–337
Attribute-Driven Design (ADD) method, 316
ASRs, 320–321
element choice, 318–319
element design solution, 321
inputs, 316
output, 317–318
repeating steps, 324
verify and refine requirements step, 321–323
Attributes. *See* Quality attributes
Audiences for documentation, 328–329
Auditor checklists, 260
Audits, 153
Authenticate actors tactic, 152
Authentication in CIA approach, 148
Authorization in CIA approach, 148
Authorize actors tactic, 152
Automated delivery in Metropolis model, 535
Automatic reallocation of IP addresses, 516
Automatic scaling, 516
Automatic teller machine (ATM) banking system, 219
Automation for testability, 171–172
AUTOSAR framework, 364
Availability
analytic model space, 259
analyzing, 255–259
broker pattern, 240
calculations, 259
CAP theorem, 523
CIA approach, 147
cloud, 521
design checklist, 96–98
detect faults tactic, 87–91
general scenario, 85–86
introduction, 79–81
planning for failure, 82–85
prevent faults tactic, 94–95
recover-from-faults tactics, 91–94
summary, 98–99
tactics overview, 87
Availability of resources tactic, 136
Availability quality attribute, 307
Availability zones, 522
Avižienis, Algirdas, 79

Back door reviews, 544–545
Back-of-the-envelope analysis, 262–264
Background of architects, 51–52
Bank application, 391–392
Base mechanisms in cloud, 509–514
Basis sets for quality attributes, 261
BDUF (Big Design Up Front) process, 278
Behavior
documenting, 351–354
element, 347
in software architecture, 6–7
Benefit in economic analysis, 441–442
Benkler, Yochai, 528
Beta testing, 372
Big bang integration, 371
Big bang models, 495–496
Big Design Up Front (BDUF) process, 278
BigTable database system, 518
Binder, Robert, 167
Binding
late, 385, 388
modifiability, 124–125
user interface, 178
Binding time category
ASRs, 293
availability, 98
interoperability, 115
modifiability, 122, 127
performance, 144
quality design, 75–76
security, 156

Binding time category, *continued*
testability, 170
usability, 182
BitTorrent networks, 221
Black-box testing, 372–373
"Blind Men and the Elephant" (Saxe), 379
Blocked time in performance, 136
Blogger website, 528
Boehm, Barry, 279, 281, 286, 288
Booch, Grady, 286
Boolean logic diagrams, 83
Bottom-up adoption, 495
Bottom-up analysis mode, 284
Bottom-up schedules, 420–421
Bound execution times tactic, 138
Bound queue sizes tactic, 139
Boundaries in ADD method, 317
Box-and-line drawings
as architectures, 6
component-and-connector views, 338
BPEL (Business Process Execution Language), 108
Brainstorming
ATAM, 410
Lightweight Architecture Evaluation, 416
QAW, 295
Branson, Richard, 443
Breadth first ADD strategy, 319
Brewer, Eric, 522
Broadcast-based publish-subscribe pattern, 229
Broker pattern
availability, 255–259
description, 210–212
weaknesses, 240–242
Brooks, Fred, 47, 419
Buley, Taylor, 147
Bureaucracy in implementation, 427
Bush, Vannevar, 397
Business cases in project life-cycle context, 46
Business context
architecture influence on, 58
architectures and business goals, 49–50
Business drivers
ATAM, 404–405
Lightweight Architecture Evaluation, 416
PALM method, 305
Business goals
ASRs from, 296–304
assessment, 469
ATAM, 402
business context, 49–50
capturing, 304
categorization, 297–299
evaluation process, 400
expressing, 299–301
general scenario, 301–303
PALM method, 305
views for, 332
Business managers, 54
Business/mission presentation in QAW, 295
Business Process Execution Language (BPEL), 108
Business process improvements as business goal, 299
Business-related architect skills, 465

C&C structures. *See* Component-and-connector (C&C) patterns and structures
Caching tactic, 139
Callbacks in Model-View-Controller pattern, 214
Calls relationship in view extraction, 384
Cancel command, 179
CAP theorem, 518, 522–523
Capture scenarios for quality attributes, 196–197
Capturing
ASRs in utility trees, 304–307
business goals, 304–307
Catastrophic failures, 82
Categorization of business goals, 297–299
CBAM. *See* Cost Benefit Analysis Method (CBAM)
Change
documenting, 355–356
modifiability. *See* Modifiability
reasoning and managing, 27–28
Change control boards, 427
Change default settings tactic, 153
Chaos Monkey, 160–161
Chaucer, Geoffrey, 459
Check-in, syncing at, 368
Choice of technology category
ASRs, 293
availability, 98
interoperability, 115
modifiability, 127
performance, 144
security, 156
testability, 170
usability, 182
CIA (confidentiality, integrity, and availability) approach, 147–148
City analogy in Metropolis model, 536

class_contains_method relationship, 384
class_is_subclass_of_class relationship, 384
Class structure, 13
Classes in testability, 167
Clements, Paul, 66
Client-server patterns, 19, 217–219
Client-side proxies, 211
Clients
  broker pattern, 211
  simulators, 265
Clone-and-own practice, 482–483
Cloud environments
  architecting in, 520–523
  availability, 521
  base mechanisms, 509–514
  database systems, 517–520
  definitions, 504–505
  deployment models, 506
  economic justification, 506–509
  equipment utilization, 508–509
  IaaS model, 515–517
  introduction, 503–504
  multi-tenancy applications, 509
  PaaS model, 517
  performance, 521
  security, 520–521
  service models, 505–506
  summary, 524
Cluster managers, 515
CMG (Computer Measurement Group), 524
Co-located teams
  Agile, 277
  coordination, 427
Cockburn, Alistair, 287
COCOMO II (COnstructive COst MOdel II)
    scale factor, 279
Code
  architecture consistency, 366–368
  design in, 364
  KSLOC, 279–281
  mapping to, 334
  security, 157
  templates, 365–367
Cohesion
  in modifiability, 121–123
  in testability, 167
Cold spares, 92, 256–259
Collaborative system of systems, 106
Collating scenarios
  CBAM, 445
  NASA ECS project, 451
COMBINATION gate symbol, 84
Combining views, 343–345
Commercial implementations of map-reduce
    patterns, 234
Common Object Request Broker Architecture
    (CORBA), 212
Communicates with relation, 237
Communication
  Agile software development, 277
  architect skills, 465
  architecture, 47
  documentation for, 329
  global development, 425
  stakeholder, 29–31
Communication diagrams for traces, 353
Communications views, 341
Community clouds, 506
Compatibility in component-and-connector
    views, 336
Compatibility quality attribute, 193
Competence
  activities, 468
  architects, 459–467
  assessment, 469–472, 474–475
  assessment goals, 469
  introduction, 459–460
  models, 476
  questions, 470, 472–474
  software architecture organizations,
      467–475
  summary, 475
Competence center patterns, 19, 238
Competence set tactic, 95
Complexity
  broker pattern, 211
  quality attributes, 71
  in testability, 167–168
Component-and-connector (C&C) patterns
    and structures, 5, 10–11
  broker, 210–212
  client-server, 217–219
  Model-View-Controller, 212–215
  peer-to-peer, 220–222
  pipe-and-filter, 215–217
  publish-subscribe, 226–229
  service-oriented architecture, 222–226
  shared-data, 230–231
  types, 13–14
  views, 335–339, 344, 406
Components, 5
  independently developed, 35–36
  replacing for testability, 167
  substituting in variation mechanism, 492
Comprehensive models for behavior docu-
    mentation, 351, 353–354

Computer Measurement Group (CMG), 524
Computer science knowledge of architects, 466
Concepts and terms, 368–369
Conceptual integrity of architecture, 189
Concrete quality attribute scenarios, 69
Concurrency
  component-and-connector views, 13–14, 337
  handling, 132–133
Condition monitoring tactic, 89
Confidence in usability, 175
Confidentiality, integrity, and availability (CIA) approach, 147–148
Configurability quality attribute, 307
Configuration manager roles, 422
Configurators, 492
Conformance, 380–381
  by analysis, 389–392
  architectural, 48
  by construction, 389
Conformance checkers, 54
Conformity Monkey, 161
Connectors
  component-and-connector views, 335–339
  multi-tier pattern, 236
  peer-to-peer systems, 220
  REST, 225
  UML, 369
Consistency
  CAP theorem, 523
  code and architecture, 366–368
  databases, 520
Consolidation in QAW, 295
Constraints
  ADD method, 322–323
  allocation views, 339
  broker pattern, 211
  client-server pattern, 218
  component-and-connector views, 337
  conformance, 390
  defining, 32–33
  layered pattern, 207
  map-reduce patterns, 235
  Model-View-Controller pattern, 213
  modular views, 333
  multi-tier pattern, 236–237
  peer-to-peer pattern, 222
  pipe-and-filter pattern, 216
  publish-subscribe pattern, 227
  requirements, 64–65
  service-oriented architecture pattern, 225
  shared-data pattern, 231
Construction, conformance by, 389
COnstructive COst MOdel II (COCOMO II) scale factor, 279
Content-based publish-subscribe pattern, 229
Contention for resources tactic, 136
Context diagrams
  ATAM presentations, 406
  in documentation, 347
Contexts
  architecture influence, 56–58
  business, 49–51, 58
  decision-making, 438–439
  professional, 51–52
  project life-cycle, 44–48
  in relationships, 204–205
  stakeholders, 52–55
  summary, 59
  technical, 40–43
  thought experiments, 263
  types, 39–40
Contextual factors in evaluation, 399–400
Continuity as business goal, 298
Control relation in map-reduce patterns, 235
Control resource demand tactic, 137–138
Control tactics for testability, 164–167
Controllers in Model-View-Controller pattern, 213–214
Conway's law, 38
Coordination model category
  ASRs, 293
  availability, 96
  global development, 426
  interoperability, 114
  modifiability, 126
  performance, 143
  quality design decisions, 73–74
  security, 155
  testability, 169
  usability, 181
CORBA (Common Object Request Broker Architecture), 212
Core asset units, 497
Core requirements, 531–532
Core vs. periphery in Metropolis model, 534
Correlation logic for faults, 81
Cost Benefit Analysis Method (CBAM), 442
  cost determination, 444
  results, 456–457
  steps, 445–447
  utility curve determination, 442–443
  variation points, 448–450
  weighting determination, 444

Costs
CBAM, 444
of change, 118
estimates, 34
global development, 423–424
independently developed components for, 36
power, 507
resources, 244
thought experiments, 263
value for, 442
Costs to complete measure, 430
Coupling
in modifiability, 121–124
in testability, 167
Crashes and availability, 85
Credit cards, 147, 157, 260, 268
Crisis, syncing at, 368
Criteria for ASRs, 306
Crowd management in Metropolis model, 534
Crowdsourcing, 528
CRUD operations, 109
CruiseControl tool, 172
Crystal Clear method, 44, 287
Cummins, Inc., 480, 490
Cunningham, Ward, 286
Customers
communication with, 29
edge-dominant systems, 529
Customization of user interface, 180

Darwin, Charles, 275
Data Access Working Group (DAWG), 451
Data accessors in shared-data pattern, 230–231
Data latency, utility trees for, 306
Data model category, 13
ASRs, 293
availability, 96
interoperability, 114
modifiability, 126
performance, 143
quality design decisions, 74
security, 155
testability, 169
usability, 182
Data reading and writing in shared-data pattern, 230–231
Data replication, 139
Data sets
map-reduce pattern, 232–233
for testability, 170–171
Data stores in shared-data pattern, 230–231
Data transformation systems, 215
Database administrators, 54
Database systems
cloud, 517–520
in reconstruction, 386–387
DataNodes, 512–514
DAWG (Data Access Working Group), 451
Deadline monotonic prioritization strategy, 140
Deadlines in processing, 134
Debugging brokers, 211
Decision makers on ATAM teams, 401
Decision-making context, 438–439
Decisions
evaluating, 398
mapping to quality requirements, 402–403
quality design, 72–76
Decomposition
description, 311–312
module, 5, 12, 16
views, 16, 343, 345
Dedicated finite resources, 530
Defects
analysis, 374
eliminating, 486
tracking, 430
Defer binding
modifiability, 124–125
user interface, 178
Degradation tactic, 93
Delegation connectors, 369
Demilitarized zones (DMZs), 152
Denial-of-service attacks, 79, 521, 533
Dependencies
basis set elements, 261
on computations, 136
intermediary tactic for, 123–124
user interface, 178
Dependent events in probability, 257
Depends-on relation
layered pattern, 207
modules, 332–333
Deploy on relation, 235
Deployability attribute, 129, 187
Deployers, 54
Deployment models for cloud, 506
Deployment structure, 14
Deployment views
ATAM presentations, 406
combining, 345
purpose, 332
Depth first ADD strategy, 319

Design and design strategy, 311
ADD. *See* Attribute-Driven Design (ADD) method
architecturally significant requirements, 311–312
in code, 364
decomposition, 311–312
early decisions, 31–32
generate and test process, 313–316
initial hypotheses, 314–315
next hypotheses, 315
quality attributes, 197
summary, 325
test choices, 315
Design checklists
availability, 96–98
design strategy hypotheses, 315
interoperability, 114–115
modifiability, 125–127
performance, 142–144
quality attributes, 199
security, 154–156
summary, 183
testability, 169–170
usability, 181–182
Designers
description and interests, 54
evaluation by, 397–398
Detect attacks tactics, 151
Detect faults tactic, 87–91
Detect intrusion tactic, 151
Detect message delay tactic, 151
Detect service denial tactic, 151
Deutsche Bank, 480
Developers
edge-dominant systems, 529
roles, 422
Development
business context, 50–51
global, 423–426
incremental, 428
project life-cycle context, 44–45
tests, 374
Development distributability attribute, 186
Deviation
failure from, 80
measuring, 429
Devices in ADD method, 317
DiNucci, Darcy, 527
dir_contains_dir relationship, 384
dir_contains_file relationship, 384
Directed system of systems, 106
Directories in documentation, 349
DiscoTect system, 391
Discover service tactic, 111
Discovery in interoperability, 105
Discovery services, 533
Distributed computing, 221
Distributed development, 427
Distributed testing in Metropolis model, 535
DMZs (demilitarized zones), 152
DNS (domain name server), 514
Doctor Monkey, 161
Documentation
Agile development projects, 356–357
architect duties, 462
architectural structures, 17–18
architecture, 47, 347–349
behavior, 351–354
changing architectures, 355–356
distributed development, 427
global development, 426
introduction, 327–328
notations, 329–331
online, 350
packages, 345–351
patterns, 350–351
and quality attributes, 354–355
services, 533
summary, 359
uses and audiences for, 328–329
views. *See* Views
YAGNI, 282
Documentation maps, 347–349
Documents, control information, 347
Domain decomposition, 315
Domain knowledge of architects, 467
Domain name server (DNS), 514
Drivers
ATAM, 404–405
Lightweight Architecture Evaluation, 416
PALM method, 305
QAW, 295
DSK (Duties, Skills, and Knowledge) model of competence, 476
Duke's Bank application, 391–392
Duties
architects, 460–464
competence, 472
professional context, 51
Duties, Skills, and Knowledge (DSK) model of competence, 476
Dynamic allocation views, 340
Dynamic analysis with fault trees, 83
Dynamic priority scheduling strategies, 140–141

Dynamic structures, 5
Dynamic system information, 385–386

Earliest-deadline-first scheduling strategy, 141
Early design decisions, 31–32
Earth Observing System Data Information System (EOSDIS) Core System (ECS). *See* NASA ECS project
eBay, 234
EC2 cloud service, 81, 160, 522, 532
Eclipse platform, 228
Economic analysis
  basis, 439–442
  benefit and normalization, 441–442
  case study, 451–457
  CBAM. *See* Cost Benefit Analysis Method (CBAM)
  cost value, 442
  decision-making context, 438–439
  introduction, 437
  scenario weighting, 441
  side effects, 441
  summary, 457
  utility-response curves, 439–441
Economics
  cloud, 506–509
  issues, 543
Economies of scale in cloud, 507–508
Ecosystems, 528–530
ECS system. *See* NASA ECS project
Edge-dominant systems, 528–530
Edison, Thomas, 203
eDonkey networks, 221
Education, documentation as, 328–329
Effective resource utilization, 187
Effectiveness category for quality, 189
Efficiency category for quality, 189–190
Einstein, Albert, 175
EJB (Enterprise Java Beans), 212
Elasticity, rapid, 504–505
Elasticity property, 187
Electric grids, 106
Electricity, 191, 570
Electronic communication in global development, 426
Elements
  ADD method, 318–319
  allocation views, 339–340
  broker pattern, 211
  catalogs, 346–347
  client-server pattern, 218
  component-and-connector views, 337
  defined, 5
  layered pattern, 207
  map-reduce patterns, 235
  mapping, 75
  Model-View-Controller pattern, 213
  modular views, 333
  multi-tier pattern, 237
  peer-to-peer pattern, 222
  pipe-and-filter pattern, 216
  product reuse, 484
  publish-subscribe pattern, 227
  service-oriented architecture pattern, 225
  shared-data pattern, 231
Employees
  as goal-object, 302
  responsibilities to, 299
Enabling quality attributes, 26–27
Encapsulation tactic, 123
Encrypt data tactic, 152
End users in edge-dominant systems, 529
Enterprise architecture vs. system architecture, 7–8
Enterprise Java Beans (EJB), 212
Enterprise resource planning (ERP) systems, 228
Enterprise service bus (ESB), 223, 225, 369
Environment
  ADD method, 317
  allocation views, 339–340
  availability, 85–86
  business goals, 300
  interoperability, 107–108
  modifiability, 119–120
  performance, 134
  quality attributes expressions, 68–70
  security, 149–150
  technical context, 41–42
  testability, 162–163
  usability, 176
  variability, 489
Environmental change as business goal, 299
ERP (enterprise resource planning) systems, 228
Errors, 80
  core handling of, 532
  detection by services, 533
  error-handling views, 341
  in usability, 175
ESB (enterprise service bus), 223, 225, 369
Escalating restart tactic, 94
Estimates, cost and schedule, 34
Evaluation
  architect duties, 462–463
  architecture, 47–48

Evaluation, *continued*
ATAM. *See* Architecture Tradeoff Analysis Method (ATAM)
contextual factors, 399–400
by designer, 397–398
Lightweight Architecture Evaluation, 415–417
outsider analysis, 399
peer review, 398–399
questions, 472
software product lines, 493–494
summary, 417
Evaluators, 54
Event bus in publish-subscribe pattern, 227
Events
Model-View-Controller pattern, 214
performance, 131, 133
probability, 257
Eventual consistency model, 168, 523
Evolutionary prototyping, 33–34
Evolving software product lines, 496–497
Exception detection tactic, 90
Exception handling tactic, 92
Exception prevention tactic, 95
Exception views, 341
Exchanging information via interfaces, 104–105
EXCLUSIVE OR gate symbol, 84
Executable assertions for system state, 166
Execution of tests, 374
Exemplar systems, 485
Exercise conclusion in PALM method, 305
Existing systems in design, 314
Expected quality attribute response levels, 453
Experience of architects, 51–52
Experiments in quality attribute modeling, 264–265
Expressing business goals, 299–301
Extensibility quality attribute, 307
Extensible programming environments, 228
Extension points for variation, 491
External sources for product lines, 496
External system representatives, 55
External systems in ADD method, 317
Externalizing change, 125
Extract-transform-load functions, 235
Extraction, raw view, 382–386
Extreme Programming development methodology, 44

Facebook, 527–528
map-reduce patterns, 234
users, 518
Fail fast principle, 522
Failure Mode, Effects, and Criticality Analysis (FMECA), 83–84
Failures, 80
availability. *See* Availability
planning for, 82–85
probabilities and effects, 84–85
Fairbanks, George, 279, 364
Fallbacks principle, 522
Fault tree analysis, 82–84
Faults, 80
correlation logic, 81
detection, 87–91
prevention, 94–95
recovery from, 91–94
Feature removal principle, 522
FIFO (first-in/first-out) queues, 140
File system managers, 516
Filters in pipe-and-filter pattern, 215–217
Financial objectives as business goal, 298
Finding violations, 389–392
Fire-and-forget information exchange, 223
Firefox, 531
First-in/first-out (FIFO) queues, 140
First principles from tactics, 72
Fixed-priority scheduling, 140
Flex software development kit, 215
Flexibility
defer binding tactic, 124
independently developed components for, 36
Flickr service, 527, 536
Flight control software, 192–193
FMECA (Failure Mode, Effects, and Criticality Analysis), 83–84
Focus on architecture in Metropolis model, 534–535
Follow-up phase in ATAM, 403–404
Folsonomy, 528
Ford, Henry, 479
Formal documentation notations, 330
Frameworks
design strategy hypotheses, 314–315
implementation, 364–365
Frankl, Viktor E., 63
Freedom from risk category for quality, 189
Functional redundancy tactic, 90
Functional requirements, 64, 66
Functional responsibility in ADD method, 322–323
Functional suitability quality attribute, 193
Functionality
component-and-connector views, 336

description, 65
Fused views, 388–389

Gamma, E., 212
Gate symbols, 83–84
General Motors product line, 487
Generalization structure, 13
Generate and test process, 313–316
Generators of variation, 492
Get method for system state, 165
Global development, 423–426
Global metrics, 429–430
Gnutella networks, 221
Goal components in business goals, 300
Goals. *See* Business goals
Goldberg, Rube, 102
Good architecture, 19–21
Good enough vs. perfect, 398
Google
database system, 518
Google App Engine, 517
Google Maps, 105–107
greenhouse gas from, 190–191
map-reduce patterns, 234
power sources, 507
Governance, 430–431
Government, responsibilities to, 299
Graceful degradation, 522
Graphical user interfaces in publish-subscribe pattern, 228
Gray-box testing, 373
Green computing, 190–191
Greenspan, Alan, 443
Growth and continuity as business goal, 298
Guerrilla movements, 543–544

Hadoop Distributed File System (HDFS), 512
Hardware costs for cloud, 507
Harel, David, 353
Harnesses for tests, 374
Hazard analysis, 82
Hazardous failures, 82
HBase database system, 518–519
HDFS (Hadoop Distributed File System), 512
Heartbeat tactic, 89, 256, 408
Helm, R., 212
Hewlett-Packard, 480
Hiatus stage in ATAM, 409
High availability. *See* Availability
Highway systems, 142
Horizontal scalability, 187
Hot spare tactic, 91
HTTP (HyperText Transfer Protocol), 219
Hudson tool, 172
Hufstedler, Shirley, 363
Human body structure, 9
Human Performance model of competence, 476
Human Performance Technology model, 469–473
Human resource management in global development, 425
Hybertsson, Henrik, 42–43
Hybrid clouds, 506
Hydroelectric power station catastrophe, 188, 192
Hypertext for documentation, 350
HyperText Transfer Protocol (HTTP), 219
Hypervisors, 510–512
Hypotheses
conformance, 390
design strategy, 314–315
fused views, 388

IaaS (Infrastructure as a Service) model, 505–506, 515–517
Identify actors tactic, 152
Ignore faulty behavior tactic, 93
Implementation, 363–364, 427
architect duties, 463
code and architecture consistency, 366–368
code templates, 365–367
design in code, 364
frameworks, 364–365
incremental development, 428
modules, 333–334
structure, 14
summary, 376
testing, 370–376
tracking progress, 428–429
tradeoffs, 427
Implementors, 55
In-service software upgrade (ISSU), 92
Includes relationship, 384
Inclusion of elements for variation, 491
Increase cohesion tactic, 123
Increase competence set tactic, 95
Increase efficiency tactic, 142
Increase resource efficiency tactic, 138
Increase resources tactic, 138–139, 142
Increase semantic coherence tactic, 123, 239
Incremental Commitment Model, 286
Incremental development, 428
Incremental integration, 371

Incremental models in adoption strategies, 495–496
Independent events in probability, 257
Independently developed components, 35–36
Inflexibility of methods, 277
Inform actors tactic, 153
Informal contacts in global development, 426
Informal notations for documentation, 330
Information handling skills, 465
Information sharing in cloud, 520
Infrastructure as a Service (IaaS) model, 505–506, 515–517
Infrastructure in map-reduce patterns, 235
Infrastructure labor costs in cloud, 507
Inheritance variation mechanism, 492
Inherits from relation, 13
INHIBIT gate symbol, 84
Inhibiting quality attributes, 26–27
Initial hypotheses in design strategy, 314–315
Inputs in ADD method, 316, 321–323
Instantiate relation, 235
Integration management in global development, 424
Integration testing, 371–372
Integrators, 55
Integrity
  architecture, 189
  CIA approach, 147
Interchangeable parts, 35–36, 480
Interfaces
  exchanging information via, 104–105
  separating, 178
Intermediary tactic, 123
Intermediate states in failures, 80
Internal sources of product lines, 496–497
Internet Protocol (IP) addresses
  automatic reallocation, 516
  overview, 514
Interoperability
  analytic model space, 259
  design checklist, 114–115
  general scenario, 107–110
  introduction, 103–106
  service-oriented architecture pattern, 224
  and standards, 112–113
  summary, 115
  tactics, 110–113
Interpersonal skills, 465
Interpolation in CBAM, 446
Interviewing stakeholders, 294–296
Introduce concurrency tactic, 139
Invokes-services role, 335
Involvement, 542–543
Iowability, 195–196
IP (Internet Protocol) addresses
  automatic reallocation, 516
  overview, 514
Is a relation, 332–333
Is-a-submodule-of relation, 12
Is an instance of relation, 13
Is part of relation
  modules, 332–333
  multi-tier pattern, 237
ISO 25010 standard, 66, 193–195
ISSU (in-service software upgrade), 92
Iterative approach
  description, 44
  reconstruction, 382
  requirements, 56

Janitor Monkey, 161
JavaScript Object Notation (JSON) form, 519
Jitter, 134
Jobs, Steve, 311
Johnson, R., 212
JSON (JavaScript Object Notation) form, 519
Just Enough Architecture (Fairbanks), 279, 364

Keys in map-reduce pattern, 232
Knowledge
  architects, 460, 466–467
  competence, 472–473
  professional context, 51
Kroc, Ray, 291
Kruchten, Philippe, 327
KSLOC (thousands of source lines of code), 279–281
Kundra, Vivek, 503

Labor availability in global development, 423
Labor costs
  cloud, 507
  global development, 423
Language, 542
Larger data sets in map-reduce patterns, 234
Late binding, 385, 388
Latency
  CAP theorem, 523
  performance, 133, 255
  queuing models for, 198–199
  utility trees for, 306
Latency Monkey, 161
Lattix tool, 387
Lawrence Livermore National Laboratory, 71
Layer bridging, 206

Layer structures, 13
Layer views in ATAM presentations, 406
Layered patterns, 19, 205–210
Layered views, 331–332
Leaders on ATAM teams, 401
Leadership skills, 464–465
Learning issues in usability, 175
Least-slack-first scheduling strategy, 141
LePatner, Barry, 3
Letterman, David, 443
Levels
  failure, 258
  restart, 94
  testing, 370–372
Leveson, Nancy, 200
Lexical analyzers, 386
Life cycle
  architecture in, 271–274
  changes, 530–531
  Metropolis model, 537
  project. *See* Project life-cycle context
  quality attribute analysis, 265–266
Life-cycle milestones, syncing at, 368
Lightweight Architecture Evaluation method, 415–417
Likelihood of change, 117
Limit access tactic, 152
Limit complexity tactic, 167
Limit event response tactic, 137
Limit exposure tactic, 152
Limit structural complexity tactic, 167–168
Linux, 531
List-based publish-subscribe pattern, 229
Load balancers, 139
Local changes, 27–28
Local knowledge of markets in global development, 423
Localize state storage for testability, 165
Locate tactic, 111
Location independence, 504
Lock computer tactic, 153
Logical threads in concurrency, 13–14

Macros for testability, 167
Mailing lists in publish-subscribe pattern, 228
Maintain multiple copies tactic, 142
Maintain multiple copies of computations tactic, 139
Maintain multiple copies of data tactic, 139
Maintain system model tactic, 180
Maintain task model tactic, 180
Maintain user model tactic, 180
Maintainability quality attribute, 195, 307
Maintainers, 55
Major failures, 82
Manage event rate tactic, 142
Manage resources tactic, 137–139
Manage sampling rate tactic
  performance, 137
  quality attributes, 72
Management and governance
  architect skills, 464
  governance, 430–431
  implementing, 427–429
  introduction, 419
  measuring, 429–430
  organizing, 422–426
  planning, 420–421
  summary, 432
Management information in modules, 334
Managers, communication with, 29
Managing interfaces tactic, 111
Manifesto for Agile software development, 276
Map architectural strategies in CBAM, 446
Map-reduce pattern, 232–235
Mapping
  to requirements, 355, 402–403
  to source code units, 334
Mapping among architectural elements category
  ASRs, 293
  availability, 97
  interoperability, 114
  modifiability, 127
  performance, 144
  quality design decisions, 75
  security, 155
  testability, 169
  usability, 182
Maps, documentation, 347–349
Market position as business goal, 299
Marketability category for quality, 190
Markov analysis, 83
Matrixed team members, 422
McGregor, John, 448
Mean time between failures (MTBF), 80, 255–259
Mean time to repair (MTTR), 80, 255–259
Measured services, 505
Measuring, 429–430
Meetings
  global development, 426
  progress tracking, 428
Methods in product reuse, 484

Metrics, 429–430
Metropolis structure
  edge-dominant systems, 528–530
  implications, 533–537
Microsoft Azure, 517
Microsoft Office 365, 509
Migrates-to relation, 14
Mill, John Stuart, 527
Minimal cut sets, 83
Minor failures, 82
Missile defense system, 104
Missile warning system, 192
Mixed initiative in usability, 177
Mobility attribute, 187
Model driven development, 45
Model-View-Controller (MVC) pattern
  overview, 212–215
  performance analysis, 252–254
  user interface, 178
Models
  product reuse, 484
  quality attributes, 197–198
  transferable and reusable, 35
Modifiability
  analytic model space, 259
  component-and-connector views, 337
  design checklist, 125–127
  general scenario, 119–120
  introduction, 117–119
  managing, 27
  ping/echo, 243
  restrict dependencies tactic, 246
  scheduling policy tactic, 244–245
  summary, 128
  tactics, 121–125
  and time-to-market, 284
  unit testing, 371
  in usability, 179
Modularity of core, 532
Modules and module patterns, 10, 205–210
  coupling, 121
  decomposition structures, 5
  description, 4–5
  types, 12–13
  views, 332–335, 406
MongoDB database, 519
Monitor relation in map-reduce patterns, 235
Monitor tactic, 88–89
Monitorability attribute, 188
MoSCoW style, 292
MSMQ product, 224
MTBF (mean time between failures), 80, 255–259
MTTR (mean time to repair), 80, 255–259
Multi-tenancy
  cloud, 509, 520
  description, 505
Multi-tier patterns, 19, 235–237
Multitasking, 132–133
Musket production, 35–36
MVC (Model-View-Controller) pattern
  overview, 212–215
  performance analysis, 252–254
  user interface, 178
Mythical Man-Month (Brooks), 47

NameNode process, 512–513
Names for modules, 333
NASA ECS project, 451
  architectural strategies, 452–456
  assign utility, 452
  collate scenarios, 451
  expected quality attribute response level, 453
  prioritizing scenarios, 452
  refining scenarios, 451–452
Nation as goal-object, 302
National Reconnaissance Office, 481
.NET platform, 212
Netflix
  cloud, 522
  Simian Army, 160–161
Network administrators, 55
Networked services, 36
Networks, cloud, 514
Nightingale application, 306–307
No effect failures, 82
Node managers, 516
Nokia, 480
non-ASR requirements, 312–313
Non-stop forwarding (NSF) tactic, 94
Nondeterminism in testability, 168
Nonlocal changes, 27
Nonrepudiation in CIA approach, 148
Nonrisks in ATAM, 402
Normalization
  databases, 520
  economic analysis, 441–442
NoSQL database systems, 518–520, 523
NoSQL movement, 248
Notations
  component-and-connector views, 338–339
  documentation, 329–331
Notifications
  failures, 80
  Model-View-Controller pattern, 214

NSF (non-stop forwarding) tactic, 94
Number of events not processed measurement, 134

Object-oriented systems
  in testability, 167
  use cases, 46
Objects in sequence diagrams, 352
Observability of failures, 80
Observe system state tactics, 164–167
Off-the-shelf components, 36
Omissions
  availability faults from, 85
  for variation, 491
On-demand self-service, 504
1+1 redundancy tactic, 91
Online documentation, 350
Ontologies, 368–369
OPC (Order Processing Center) component, 224, 226
Open content systems, 529
Open Group
  certification program, 477
  governance responsibilities, 430–431
Open source software, 36, 238
Operation Desert Storm, 104
OR gate symbol, 84
Orchestrate tactic, 111
Orchestration servers, 223, 225
Order Processing Center (OPC) component, 224, 226
Organization
  global development, 423–426
  project manager and software architect responsibilities, 422–423
  software development teams, 422
Organizational Coordination model, 470, 473, 476
Organizational Learning model, 470, 474, 476
Organizations
  activities for success, 468
  architect skills, 464
  architecture influence on, 33
  as goal-object, 302
  security processes, 157
  structural strategies for products, 497
Outages. *See* Availability
Outputs
  ADD method, 317–318
  ATAM, 402–403
Outsider analysis, 399
Overlay views, 343
Overloading for variation, 491
Overview presentations in PALM method, 305

P2P (peer-to-peer) pattern, 220–222
PaaS (Platform as a Service) model, 505, 517
Page mappers, 510–512
PALM (Pedigreed Attribute eLicitation Method), 304–305
Parameter fence tactic, 90
Parameter typing tactic, 90
Parameters for variation mechanism, 492
Parser tool, 386
Partitioning CAP theorem, 523
Partnership and preparation phase in ATAM, 403–404
Passive redundancy, 91–92, 256–259
Patterns, 18–19
  allocation, 232–237
  component-and-connector. *See* Component-and-connector (C&C) patterns and structures
  documenting, 350–351
  introduction, 203–204
  module, 205–210
  relationships, 204–205
  summary, 247–248
  and tactics, 238–247, 315
Paulish, Dan, 420
Pause/resume command, 179
Payment Card Industry (PCI), 260
PDF (probability density function), 255
PDM (platform-definition model), 45
Pedigree and value component of business goals, 301
Pedigreed Attribute eLicitation Method (PALM), 304–305
Peer nodes, 220
Peer review, 398–399
Peer-to-peer (P2P) pattern, 220–222
Penalties in Incremental Commitment Model, 286
People
  managing, 464
  in product reuse, 485
Perfect vs. good enough, 398
Performance
  analytic model space, 259
  analyzing, 252–255
  broker pattern, 241
  cloud, 521
  component-and-connector views, 336
  control resource demand tactics, 137–138
  design checklist, 142–144

Performance, *continued*
general scenario, 132–134
introduction, 131–132
manage resources tactics, 138–139
map-reduce pattern, 232
ping/echo, 243
and quality, 191
quality attributes tactics, 72
queuing models for, 198–199
resource effects, 244, 246
summary, 145
tactics overview, 135–137
views, 341
Performance quality attribute, 307
Performance efficiency quality attribute, 193
Periodic events, 133
Periphery
Metropolis model, 535
requirements, 532
Persistent object managers, 515–516
Personal objectives as business goal, 298
Personnel availability in ADD method, 320
Petrov, Stanislav Yevgrafovich, 192
Phases
ATAM, 403–404
metrics, 430
Metropolis model, 534
Philips product lines, 480–481, 487
Physical security, 191
PIM (platform-independent model), 45
Ping/echo tactic, 87–88, 243
Pipe-and-filter pattern, 215–217
Planned increments, 530
Planning
for failure, 82–85
incremental development, 428
overview, 420–421
tests, 374
Platform as a Service (PaaS) model, 505, 517
Platform-definition model (PDM), 45
Platform-independent model (PIM), 45
Platforms
architect knowledge about, 467
frameworks in, 365
patterns, 19, 238
services for, 532–533
Plug-in architectures, 34
PMBOK (Project Management Body of Knowledge), 423–425
Pointers, smart, 95
Policies, scheduling, 140
Pooling resources, 504
Portability quality attributes, 67, 186, 195
Portfolio as goal-object, 302
Ports in component-and-connector views, 335, 337–338
Potential alternatives, 398
Potential problems, peer review for, 399
Potential quality attributes, 305
Power station catastrophe, 188, 192
Predicting system qualities, 28
Predictive model tactic, 95
Preemptible processes, 141
Preparation-and-repair tactic, 91–93
Preprocessor macros, 167
Presentation
ATAM, 402–406
documentation, 346
Lightweight Architecture Evaluation, 416
PALM method, 305
QAW, 295
Prevent faults tactics, 94–95
Primary presentations in documentation, 346
Principles
Agile, 276–277
cloud failures, 522
design fragments from, 72
Incremental Commitment Model, 286
Prioritize events tactic, 137–138, 142
Prioritizing
ATAM scenarios, 410
CBAM scenarios, 445–446
CBAM weighting, 444
Lightweight Architecture Evaluation scenarios, 416
NASA ECS project scenarios, 452
QAW, 295–296
risk, 429
schedules, 140–141
views, 343
PRIORITY AND gate symbol, 84
Private clouds, 506
Private IP addresses, 514
Proactive enforcement in Metropolis model, 535
Proactive product line models, 495
Probability density function (PDF), 255
Probability for availability, 256–259
Problem relationships in patterns, 204–205
Proceedings scribes, 401
Processes
development, 44–45
product reuse, 484
recommendations, 20
security, 157
Processing time in performance, 136

Procurement management, 425
Product-line managers, 55
Product lines. *See* Software product lines
Product manager roles, 422
Productivity metrics, 429–430
Professional context, 51–52, 58
Profiler tools, 386
Programming knowledge of architects, 466
Project context, 57
Project life-cycle context
  architecturally significant requirements, 46–47
  architecture analysis and evaluation, 47–48
  architecture documentation and communication, 47
  architecture selection, 47
  business cases, 46
  development processes, 44–45
  implementation conformance, 48
Project Management Body of Knowledge (PMBOK), 423–425
Project managers
  description and interests, 55
  responsibilities, 422–423
Project planning artifacts in product reuse, 484
Propagation costs of change, 288
Prosumers in edge-dominant systems, 529
Protection groups, 91
Prototypes
  evolutionary, 33–34
  quality attribute modeling and analysis, 264–265
  for requirements, 47
Provides-services role, 335
Proxy servers, 146, 211
Public clouds, 506
Public IP addresses, 514
Publicly available apps, 36
Publish-subscribe connector, 336
Publish-subscribe pattern, 226–229
Publisher role, 336

QAW (Quality Attribute Workshop), 294–296
Qt framework, 215
Quality attribute modeling and analysis, 251–252
  analytic model space, 259–260
  availability analysis, 255–259
  checklists, 260–262
  experiments, simulations, and prototypes, 264–265
  life cycle stages, 265–266
  performance analysis, 252–255
  summary, 266–267
  thought experiments and back-of-the-envelope analysis, 262–264
Quality Attribute Workshop (QAW), 294–296
Quality attributes, 185
  ADD method, 322–323
  ASRs, 294–296
  ATAM, 407
  capture scenarios, 196–197
  categories, 189–190
  checklists, 199, 260–262
  considerations, 65–67
  design approaches, 197
  and documentation, 354–355
  grand unified theory, 261
  important, 185–188
  inhibiting and enabling, 26–27
  introduction, 63–64
  Lightweight Architecture Evaluation, 416
  models, 197–198
  NASA ECS project, 453
  peer review, 398
  quality design decisions, 72–76
  requirements, 64, 68–70
  software and system, 190–193
  standard lists, 193–196
  summary, 76–77
  tactics, 70–72, 198–199
  technical context, 40–41
  variability, 488–489
  X-ability, 196–199
Quality design decisions, 72–73
  allocation of responsibilities, 73
  binding time, 75–76
  coordination models, 73–74
  data models, 74
  element mapping, 75
  resource management, 74–75
  technology choices, 76
Quality management in global development, 424
Quality of products as business goal, 299
Quality requirements, mapping decisions to, 402–403
Quality views, 340–341
Questioners on ATAM teams, 401
Questions for organizational competence, 470, 472–474
Queue sizes tactic, 139
Queuing models for performance, 198–199, 252–255
Quick Test Pro tool, 172

Race conditions, 133
Random access in equipment utilization, 508
Rapid elasticity, 504–505
Rate monotonic prioritization strategy, 140
Rational Unified Process, 44
Rationale in documentation, 347, 349
Raw view extraction in reconstruction, 382–386
RDBMSs (relational database management systems), 518
React to attacks tactics, 153
Reactive enforcement in Metropolis model, 536
Reactive product line models, 495
Reader role in component-and-connector views, 335
Reconfiguration tactic, 93
Reconstruction and conformance, 380–381
  database construction, 386–387
  finding violations, 389–392
  guidelines, 392–393
  process, 381–382
  raw view extraction, 382–386
  summary, 393–394
  view fusion, 388–389
Record/playback method for system state, 165
Recover from attacks tactics, 153–154
Recover-from-faults tactics, 91–94
Reduce computational overhead tactic, 142
Reduce function in map-reduce pattern, 232–235
Reduce overhead tactic, 138
Redundancy tactics, 90, 256–259
Refactor tactic, 124
Refined scenarios
  NASA ECS project, 451–452
  QAW, 296
Reflection for variation, 491
Reflection pattern, 262
Registry of services, 225
Regression testing, 372
Reintroduction tactics, 91, 93–94
Rejuvenation tactic, 95
Relational database management systems (RDBMSs), 518
Relations
  allocation views, 339–340
  architectural structures, 14, 16–17
  broker pattern, 211
  client-server pattern, 218
  component-and-connector views, 337
  conformance, 390
  in documentation, 346
  layered pattern, 207
  map-reduce patterns, 235
  Model-View-Controller pattern, 213
  modular views, 333
  multi-tier pattern, 237
  peer-to-peer pattern, 222
  pipe-and-filter pattern, 216
  publish-subscribe pattern, 227
  service-oriented architecture pattern, 225
  shared-data pattern, 231
  view extraction, 384
Release strategy for documentation, 350
Reliability
  cloud, 507
  component-and-connector views, 336
  core, 532
  independently developed components for, 36
  vs. safety, 188
  SOAP, 109
  views, 341
Reliability quality attribute, 195
Remote procedure call (RPC) model, 109
Removal from service tactic, 94–95
Replicated elements in variation, 491
Replication tactic, 90
Report method for system state, 165
Reporting tests, 374
Repository patterns, 19
Representation of architecture, 6
Representational State Transfer (REST), 108–110, 223–225
Reputation of products as business goal, 299
Request/reply connectors
  client-server pattern, 218
  peer-to-peer pattern, 222
Requirements
  ASRs. *See* Architecturally significant requirements (ASRs)
  categories, 64–65
  from goals, 49
  mapping to, 355, 402–403
  Metropolis model, 534
  product reuse, 483
  prototypes for, 47
  quality attributes, 68–70
  software development life cycle changes, 530
  summary, 308–310
  tying methods together, 308
Requirements documents
  ASRs from, 292–293

Waterfall model, 56
Reset method for system state, 165
Resisting attacks tactics, 152–153
RESL scale factor, 279
Resource management category
ASRs, 293
availability, 97
interoperability, 115
modifiability, 127
performance, 144
quality design decisions, 74–75
security, 155
software development life cycle changes, 530
testability, 170
usability, 182
Resources
component-and-connector views, 336
equipment utilization, 508
pooling, 504
sandboxing, 166
software development life cycle changes, 530
Response
availability, 85–86
interoperability, 105, 107–108
modifiability, 119–120
performance, 134
quality attributes expressions, 68–70
security, 149–150
testability, 162–163
usability, 176
variability, 489
Response measure
availability, 85–86
interoperability, 107–108
modifiability, 119–120
performance, 134
quality attributes expressions, 68–70
security, 149–150
testability, 162–163
usability, 176
variability, 489
Responsibilities
as business goal, 299
modules, 333
quality design decisions, 73
REST (Representational State Transfer), 108–110, 223–225
Restart tactic, 94
Restrict dependencies tactic, 124, 239, 246–247
Restrictions on vocabulary, 36
Results
ATAM, 411
CBAM, 447, 456–457
evaluation, 400
Lightweight Architecture Evaluation, 416
Retry tactic, 93
Reusable models, 35
Reuse of software architecture, 479, 483–486
Reviews
back door, 544–545
peer, 398–399
Revision history of modules, 334
Revoke access tactic, 153
Rework in agility, 279
Risk
ADD method, 320
ATAM, 402
global development, 425
progress tracking, 429
Risk-based testing, 373–374
Robustness of core, 532
Roles
component-and-connector views, 335
product line architecture, 488–490
software development teams, 422
testing, 375–376
Rollback tactic, 92
Round-robin scheduling strategy, 140–141
Rozanski, Nick, 170
RPC (remote procedure call) model, 109
Runtime conditionals, 492
Rutan, Burt, 159

SaaS (Software as a Service) model, 505
Safety
checklists, 260, 268
use cases, 46
Safety attribute, 188
Safety Integrity Level, 268
Salesforce.com, 509
Sample technologies in cloud, 514–520
Sampling rate tactic, 137
Sandbox tactic, 165–166
Sanity checking tactic, 89
Satisfaction in usability, 175
Saxe, John Godfrey, 379
Scalability
kinds, 187
peer-to-peer systems, 220
WebArrow web-conferencing system, 285
Scalability attribute, 187
Scaling, automatic, 516
Scenario scribes, 401

Scenarios
ATAM, 408, 410
availability, 85–86
business goals, 301–303
CBAM, 445–446
interoperability, 107–110
Lightweight Architecture Evaluation, 416
modifiability, 119–120
NASA ECS project, 451–452
performance, 132–134
QAW, 295–296
quality attributes, 67–70, 196–197
security, 148–150
for structures, 12
testability, 162–163
usability, 176
weighting, 441, 444
Schedule resources tactic
performance, 139
quality attributes, 72
Scheduled downtimes, 81
Schedulers, hypervisor, 512
Schedules
deviation measurements, 429
estimates, 34
policies, 140–141
policy tactic, 244–245
top-down and bottom-up, 420–421
Schemas, database, 519
Scope, product line, 486–488
Scope and summary section in documentation maps, 347
Scrum development methodology, 44
SDL (Specification and Description Language), 354
Security
analytic model space, 259
broker pattern, 242
cloud, 507, 520–521
component-and-connector views, 336
design checklist, 154–156
general scenario, 148–150
introduction, 147–148
ping/echo, 243
quality attributes checklists, 260
summary, 156
tactics, 150–154
views, 341
Security Monkey, 161
Security quality attribute, 195, 307
SEI (Software Engineering Institute), 59
Selecting
architecture, 47
tools and technology, 463
Selenium tool, 172
Self-organization in Agile, 277
Self-test tactic, 91
Semantic coherence, 178
Semantic importance, 140
Semiformal documentation notations, 330
Sensitivity points in ATAM, 403
Separate entities tactic, 153
Separation of concerns in testability, 167
Sequence diagrams
thought experiments, 263
for traces, 351–352
Servers
client-server pattern, 217–219
proxy, 146, 211
SAO pattern, 223, 225
Service consumer components, 222, 225
Service discovery in SOAP, 108
Service impact of faults, 81
Service-level agreements (SLAs)
Amazon, 81, 522
availability in, 81
IaaS, 506
PaaS, 505
SOA, 222
Service-oriented architecture (SOA) pattern, 222–226
Service providers, 222–225
Service registry, 223
Service structure, 13
Services for platforms, 532–533
Set method for system state, 165
Shadow tactic, 93
Shared-data patterns, 19, 230–231
Shared documents in documentation, 350
Shareholders, responsibilities to, 299
Siberian hydroelectric plant catastrophe, 188, 192
Siddhartha, Gautama, 251
Side-channel attacks, 521
Side effects in economic analysis, 439, 441
Simian Army, 160–161
Simulations, 264–265
Size
modules, 121
queue, 139
Skeletal systems, 34
Skeletal view of human body, 9
Skills
architects, 460, 463, 465
global development, 423
professional context, 51

SLAs. *See* Service-level agreements (SLAs)
Small victories, 544
Smart pointers, 95
SOA (service-oriented architecture) pattern, 222–226
SOAP
  vs. REST, 108–110
  SOA pattern, 223–225
Social networks in publish-subscribe pattern, 229
Socializing in Incremental Commitment Model, 286
Society
  as goal-object, 302
  service to, 299
Software architecture importance, 25–26
  change management, 27–28
  constraints, 32–33
  cost and schedule estimates, 34
  design decisions, 31–32
  evolutionary prototyping, 33–34
  independently developed components, 35–36
  organizational structure, 33
  quality attributes, 26–27
  stakeholder communication, 29–31
  summary, 37
  system qualities prediction, 28
  training basis, 37
  transferable, reusable models, 35
  vocabulary restrictions, 36
Software architecture overview, 3–4. *See also* Architecture
  as abstraction, 5–6
  behavior in, 6–7
  competence, 467–475
  contexts. *See* Contexts
  definitions, 4
  good and bad, 19–21
  patterns, 18–19
  selecting, 7
  as set of software structures, 4–5
  structures and views, 9–18
  summary, 21–22
  system architecture vs. enterprise, 7–8
Software as a Service (SaaS) model, 505
Software Engineering Body of Knowledge (SWEBOK), 292
Software Engineering Institute (SEI), 59, 479
Software Product Line Conference (SPLC), 498
Software Product Line Hall of Fame, 498
Software product lines
  adoption strategies, 494–496
  evaluating, 493–494
  evolving, 496–497
  failures, 481–482
  introduction, 479–481
  key issues, 494–497
  organizational structure, 497
  quality attribute of variability, 488
  reuse potential, 483–486
  role of, 488–490
  scope, 486–488
  successful, 483–486
  summary, 497–498
  variability, 482–483
  variation mechanisms, 490–493
Software quality attributes, 190–193
Software rejuvenation tactic, 95
Software upgrade tactic, 92–93
Solutions in relationships, 204–205
SonarJ tool, 387–391
Sorting in map-reduce pattern, 232
SoS (system of systems), 106
Source code
  KSLOC, 279–281
  mapping to, 334
Source in security scenario, 150
Source of stimulus
  availability, 85–86
  interoperability, 107–108
  modifiability, 119–120
  performance, 134
  quality attributes expressions, 68–70
  security, 148
  testability, 162–163
  usability, 176
  variability, 489
Spare tactics, 91–92, 256–259
Specialized interfaces tactic, 165
Specification and Description Language (SDL), 354
Spikes in Agile, 284–285
SPLC (Software Product Line Conference), 498
Split module tactic, 123
Sporadic events, 133
Spring framework, 166
Staging views, 343
Stakeholders
  on ATAM teams, 401
  communication among, 29–31, 329
  documentation for, 348–349
  evaluation process, 400
  interests, 52–55
  interviewing, 294–296

Stakeholders, *continued*
for methods, 272
utility tree reviews, 306
views, 342
Standard lists for quality attributes, 193–196
Standards and interoperability, 112–113
State, system, 164–167
State machine diagrams, 353
State resynchronization tactic, 93
Stateless services in cloud, 522
States, responsibilities to, 299
Static allocation views, 340
Static scheduling, 141
Status meetings, 428
Stein, Gertrude, 142
Steinberg, Saul, 39
Stimulus
availability, 85–86
interoperability, 107–108
modifiability, 119–120
performance, 134
quality attributes expressions, 68–70
security, 148, 150
source. *See* Source of stimulus
testability, 162–163
usability, 176
variability, 489
Stochastic events, 133
Stonebraker, Michael, 518
Storage
for testability, 165
virtualization, 512–513
Strategies in NASA ECS project, 452–456
Strictly layered patterns, 19
Structural complexity in testability, 167–168
Structure101 tool, 387
Stuxnet virus, 80
Subarchitecture in component-and-connector views, 335
Submodules, 333
Subscriber role, 336
Subsystems, 9
Supernodes in peer-to-peer pattern, 220
Support and development software, 358–359
Support system initiative tactic, 180–181
Support user initiative tactic, 179–180
SWEBOK (Software Engineering Body of Knowledge), 292
Swing classes, 215
Syncing code and architecture, 368
System analysis and construction, documentation for, 329
System architecture vs. enterprise architecture, 7–8
System as goal-object, 302
System availability requirements, 81
System efficiency in usability, 175
System engineers, 55
System exceptions tactic, 90
System Generation Module, 358
System initiative in usability, 177
System of systems (SoS), 106
System overview in documentation, 349
System qualities, predicting, 28
System quality attributes, 190–193
System test manager roles, 422
System testing, 371

Tactics
availability, 87–96
interactions, 242–247
interoperability, 110–113
modifiability, 121–125
patterns relationships with, 238–242
performance, 135–142
quality attributes, 70–72, 198–199
security, 150–154
testability, 164–168
usability, 177–181
Tailor interface tactic, 111
Team building skills, 463, 465
Team leader roles, 422
TeamCity tool, 172
Teams
ATAM, 400–401
organizing, 422
Technical contexts
architecture influence, 57
environment, 41–42
quality attributes, 40–41
Vasa ship, 42–43
Technical debt, 286
Technical processes in security, 157
Technology choices, 76
Technology knowledge of architects, 467
Templates
ATAM, 406
code, 365–367
scenarios. *See* Scenarios
variation mechanism, 492
10-18 Monkey, 161
Terminating generate and test process, 316
Terms and concepts, 368–369
Test harnesses, 160

Testability
analytic model space, 259
automation, 171–172
broker pattern, 241
design checklist, 169–170
general scenario, 162–163
introduction, 159–162
summary, 172
tactics, 164–168
test data, 170–171
Testable requirements, 292
TestComplete tool, 172
Testers, 55
Tests and testing
activities, 374–375
architect role, 375–376, 463
black-box and white-box, 372–373
choices, 315
in incremental development, 428
levels, 370–372
modules, 334
product reuse, 484
risk-based, 373–374
summary, 376
Therac-25 fatal overdose, 192
Thought experiments, 262–264
Thousands of source lines of code (KSLOC), 279–281
Threads in concurrency, 132–133
Throughput of systems, 134
Tiers
component-and-connector views, 337
multi-tier pattern, 235–237
Time and time management
basis sets, 261
global development, 424
performance, 131
Time boxing, 264
Time of day factor in equipment utilization, 508
Time of year factor in equipment utilization, 508
Time-sharing, 503
Time stamp tactic, 89
Time to market
independently developed components for, 36
and modifiability, 284
Timeout tactic, 91
Timing in availability, 85
TMR (triple modular redundancy), 89
Tools
for product reuse, 484
selecting, 463
Top-down adoption, 495
Top-down analysis mode, 284
Top-down schedules, 420–421
Topic-based publish-subscribe patterns, 229
Topological constraints, 236
Torvalds, Linus, 530, 535, 538
Total benefit in CBAM, 446
Traces for behavior documentation, 351–353
Tracking progress, 428–429
Tradeoffs
ATAM, 403
implementation, 427
Traffic systems, 142
Training, architecture for, 37
Transactions
availability, 95
databases, 519–520
SOAP, 108
Transferable models, 35
Transformation systems, 215
Transforming existing systems, 462
Transitions in state machine diagrams, 354
Triple modular redundancy (TMR), 89
Troeh, Eve, 190
Turner, R., 279, 281, 288
Twitter, 528
Two-phase commits, 95

Ubiquitous network access, 504
UDDI (Universal Description, Discovery and Integration) language, 108
UML
activity diagrams, 353
communication diagrams, 353
component-and-connector views, 338–339
connectors, 369
sequence diagrams, 351–352
state machine diagrams, 353
Unambiguous requirements, 292
Uncertainty in equipment utilization, 508–509
Undo command, 179
Unified Process, 44
Unit testing, 370–371
Unity of purpose in modules, 121
Universal Description, Discovery and Integration (UDDI) language, 108
Up-front planning vs. agility, 278–281
Usability
analytic model space, 259
design checklist, 181–182
general scenario, 176

Usability, *continued*
introduction, 175
quality attributes checklists, 260
tactics, 177–181
Usability quality attribute, 193, 307
Usage
allocation views, 339
component-and-connector views, 337
modular views, 333
Use an intermediary tactic, 245
modifiability, 123
quality attributes, 72
Use cases
ATAM presentations, 406
thought experiments, 263
for traces, 351
"User beware" proviso, 372
User initiative in usability, 177
User interface
exchanging information via, 104–105
separating, 178
User needs in usability, 175
User stories in Agile, 278
Users
communication with, 29
description and interests, 55
Uses
for documentation, 328–329
views for, 332
Uses relation in layered patterns, 19
Uses structure in decomposition, 12
Utility
assigning, 452
CBAM, 448
Utility-response curves, 439–443
Utility trees
ASRs, 304–307
ATAM, 407, 410
Lightweight Architecture Evaluation, 416
Utilization of equipment in cloud, 508–509

Value component
business goals, 301
utility trees, 306
Value for cost (VFC), 438, 442
Variability
product line, 482–483
quality attributes, 488–489
Variability attribute, 186
Variability guides, 347, 493
Variation
binding time, 75
software product lines, 490–493
Variation points
CBAM, 448–450
identifying, 490
Vasa ship, 42–43
Vascular view of human body, 9
Vehicle cruise control systems, 353
Verify and refine requirements in ADD, 321–323
Verify message integrity tactic, 151
Vertical scalability, 187
VFC (value for cost), 438, 442
Views, 331–332
allocation, 339–340
architectural structures, 9–10
choosing, 341–343
combining, 343–345
component-and-connector, 335–339, 344, 406
documenting, 345–347
fused, 388–389
Model-View-Controller pattern, 213–214
module, 332–335, 406
quality, 340–341
Views and Beyond approach, 282, 356–357
Villa, Pancho, 541
Violations, finding, 389–392
Virtual resource managers, 515
Virtual system of systems, 106
Virtualization and virtual machines
cloud, 509–514, 520–521
layers as, 13
in sandboxing, 166
Visibility of interfaces, 333
Vitruvius, 459
Vlissides, J., 212
Vocabulary
quality attributes, 67
restrictions, 36
Voting tactic, 89
Vulnerabilities in security views, 341

Walking skeleton method, 287
War ship example, 42–43
Warm spare tactic, 91–92
Watchdogs, 89
Waterfall model
description, 44
requirements documents, 56
Weaknesses
broker pattern, 211, 240–242
client-server pattern, 218
layered pattern, 207
map-reduce patterns, 235

Model-View-Controller pattern, 213
multi-tier pattern, 237
peer-to-peer pattern, 222
pipe-and-filter pattern, 216
publish-subscribe pattern, 227
service-oriented architecture pattern, 225
shared-data pattern, 231
Wealth of Networks (Benkler), 528
Web 2.0 movement, 527
Web-based system events, 131
Web-conferencing systems
Agile example, 283–285
considerations, 265
Web Services Description Language (WSDL), 110
WebArrow web-conferencing system, 284–285
WebSphere MQ product, 224
Weighting scenarios, 441, 444
Wells, H. G., 117
West, Mae, 131
"What if" questions in performance analysis, 255
White-box testing, 372–373
Whitney, Eli, 35–36, 480
Wikipedia, 528
Wikis for documentation, 350
Wisdom of crowds, 537
Woods, Eoin, 25, 170
Work assignment structures, 14
Work-breakdown structures, 33
Work skills of architect, 465
World Wide Web as client-server pattern, 219
Wrappers, 129
Writer role in component-and-connector views, 335
WS*, 108–110
WSDL (Web Services Description Language), 110

X-ability, 196–199
X-ray view of human body, 9

YAGNI principle, 282
Yahoo! map-reduce patterns, 234
Young, Toby, 39
YouTube, 528

Zoning policies analogy in Metropolis model, 536