A Practical Guide to Distributed Scrum

By Elizabeth Woodward, Steffan Surdek, and Matthew Ganis

This is the first comprehensive, practical guide for Scrum practitioners working in large-scale distributed environments. Written by three of IBM’s leading Scrum practitioners—in close collaboration with the IBM QSE Scrum Community of more than 1,000 members worldwide—this book offers specific, actionable guidance for everyone who wants to succeed with Scrum in the enterprise.

Readers will follow a journey through the lifecycle of a distributed Scrum project, from envisioning products and setting up teams to preparing for Sprint planning and running retrospectives. Using real-world examples, the book demonstrates how to apply key Scrum practices, such as look-ahead planning in geographically distributed environments. Readers will also gain valuable new insights into the agile management of complex problem and technical domains.

Disciplined Agile Delivery

A Practitioner’s Guide to Agile Software Delivery in the Enterprise

By Scott W. Ambler and Mark Lines

It is widely recognized that moving from traditional to agile approaches to build software solutions is a critical source of competitive advantage. Mainstream agile approaches that are indeed suitable for small projects require significant tailoring for larger, complex enterprise projects. In Disciplined Agile Delivery, Scott W. Ambler and Mark Lines introduce IBM®’s breakthrough Disciplined Agile Delivery (DAD) process framework, which describes how to do this tailoring. DAD applies a more disciplined approach to agile development by acknowledging and dealing with the realities and complexities of a portfolio of interdependent program initiatives.

Ambler and Lines show how to extend Scrum with supplementary agile and lean strategies from Agile Modeling (AM), Extreme Programming (XP), Kanban, Unified Process (UP), and other proven methods to provide a hybrid approach that is adaptable to your organization’s unique needs.

Sign up for the monthly IBM Press newsletter at ibmpressbooks.com/newsletters
Being Agile
Eleven Breakthrough Techniques to Keep You from “Waterfalling Backward”
By Leslie Ekas, Scott Will

When agile teams don’t get immediate results, it’s tempting for them to fall back into old habits that make success even less likely. In Being Agile, Leslie Ekas and Scott Will present eleven powerful techniques for rapidly gaining substantial value from agile, making agile methods stick, and launching a “virtuous circle” of continuous improvement.

Ekas and Will help you clear away silos, improve stakeholder interaction, eliminate waste and waterfall-style inefficiencies, and lead the agile transition far more successfully. Each of their eleven principles can stand on its own: When you combine them, they become even more valuable.

Patterns of Information Management
By Mandy Chessell and Harald Smith

Use Best Practice Patterns to Understand and Architect Manageable, Efficient Information Supply Chains That Help You Leverage All Your Data and Knowledge

In the era of “Big Data,” information pervades every aspect of the organization. Therefore, architecting and managing it is a multi-disciplinary task. Now, two pioneering IBM® architects present proven architecture patterns that fully reflect this reality. Using their pattern language, you can accurately characterize the information issues associated with your own systems, and design solutions that succeed over both the short- and long-term.
Related Books of Interest

**Common Information Models for an Open, Analytical, and Agile World**

By Mandy Chessell, Gandhi Sivakumar, Dan Wolfson, Kerard Hogg, Ray Harishankar

Maximize the Value of Your Information Throughout Even the Most Complex IT Project

Five senior IBM architects show you how to use information-centric views to give data a central role in project design and delivery. Using Common Information Models (CIM), you learn how to standardize the way you represent information, making it easier to design, deploy, and evolve even the most complex systems.

Using a complete case study, the authors explain what CIMs are, how to build them, and how to maintain them. You learn how to clarify the structure, meaning, and intent of any information you may exchange, and then use your CIM to improve integration, collaboration, and agility.

In today’s mobile, cloud, and analytics environments, your information is more valuable than ever. To build systems that make the most of it, start right here.

**Implementing the IBM® Rational Unified Process® and Solutions**

A Guide to Improving Your Software Development Capability and Maturity

Joshua Barnes

**Software Test Engineering with IBM Rational Functional Tester**

The Definitive Resource

Davis, Chirillo, Gouveia, Saracevic, Bocarsley, Quesada, Thomas, van Lint

**Enterprise Master Data Management**

An SOA Approach to Managing Core Information

Dreibelbis, Hechler, Milman, Oberhofer, van Run, Wolfson

**An Introduction to IMS**


Barbara Klein, et al.

**Outside-in Software Development**

A Practical Approach to Building Successful Stakeholder-based Products

Carl Kessler, John Switzer

Sign up for the monthly IBM Press newsletter at ibmpressbooks.com/newsletters
This page intentionally left blank
Practical Software Architecture
Dedication

I dedicate this book to my late father, Sri. Dibakar Mitra (1940–2015). My father left us earlier this year (2015) and has left a traumatic lacuna in my life, which I find increasingly hard to deal with and to accept its veracity. Baba (father) was my ultimate motivation in life—to believe in myself and go that extra mile to achieve anything to make him immensely proud of his only son—and proud he was! He used to carry my (not even his own) business card in his wallet and show it with immense amour-propre in his professional and personal circles.

Baba left us just 45 days shy of my becoming a Distinguished Engineer at IBM®, an honor which he so desperately wanted to see happen; it remains as my single greatest regret that I could not pick up the phone and give him the news. His last words to me on his death bed were “Do not worry; your DE will happen this year.” He was put on the ventilator shortly thereafter. He had fought so hard to not leave us but had to fall victim to some utter medical negligence and incompetency of one of the so-called best hospitals in Kolkata, India (my native place); the emotional rage inside me will never cease to burn.

Baba, I hope you are at peace wherever you are, and I pray that I can only serve you in some form in my remaining lifetime. Accept my love, forever.
# Contents

Foreword ......................................................... xv

Preface ......................................................... xvi

Chapter 1  Case Study ................................................. 1
  The Business Problem .............................................. 1
  Summary .......................................................... 5

Chapter 2  Software Architecture: The What and Why ............... 7
  Some Background ................................................... 7
  The What ........................................................... 8
  The Why ........................................................... 10
  Architecture Views and Viewpoints ................................. 14
  Summary .......................................................... 17
  References ......................................................... 18

Chapter 3  Capturing Just Enough ................................... 19
  Architecture Aspects in Focus .................................... 19
  Summary .......................................................... 21

Chapter 4  The System Context ....................................... 23
  The Business Context Versus System Context Conundrum ........... 23
  Capturing the System Context ...................................... 25
  Case Study: System Context for Elixir ............................. 30
  Summary .......................................................... 36
  References ......................................................... 37
Chapter 5  The Architecture Overview ................. 39
  What It Is .................................................. 39
  Why We Need It ........................................... 41
  The Enterprise View ....................................... 42
  The Layered View .......................................... 47
  The IT System View ....................................... 52
  Case Study: Architecture Overview of Elixir ............ 57
  Summary ..................................................... 63
  References .................................................. 63

Chapter 6  Architecture Decisions ...................... 65
  Why We Need It ........................................... 65
  How to Get Started ........................................ 66
  Creating an Architecture Decision ......................... 67
  Case Study: Architecture Decisions for Elixir ............ 72
  Summary ..................................................... 75

Chapter 7  The Functional Model ....................... 77
  Why We Need It ........................................... 77
  A Few Words on Traceability ............................. 79
  Developing the Functional Model ......................... 81
  Case Study: Functional Model for Elixir ................. 99
  Summary ..................................................... 107
  References .................................................. 108

Chapter 8  The Operational Model ..................... 109
  Why We Need It ........................................... 110
  On Traceability and Service Levels ....................... 111
  Developing the Operational Model ....................... 113
  Case Study: Operational Model for Elixir ............... 141
  Summary ..................................................... 149
  References .................................................. 150

Chapter 9  Integration: Approaches and Patterns ....... 151
  Why We Need It ........................................... 151
  Approaches to Integration ................................ 152
  Integration Patterns ...................................... 161
  Case Study: Integration View of Elixir ................... 166
  Summary ..................................................... 169
  References .................................................. 170
Chapter 10  Infrastructure Matters ................................. 171
  Why We Need It .................................................. 172
  Some Considerations .............................................. 172
  Case Study: Infrastructure Considerations for Elixir ................. 192
  Summary ............................................................ 194
  So Where Do We Stand? ........................................... 195
  References .......................................................... 196

Chapter 11  Analytics: An Architecture Introduction ............... 199
  Why We Need It .................................................. 200
  Dimensions of Analytics ......................................... 201
  Analytics Architecture: Foundation ................................ 205
  Architecture Building Blocks .................................... 216
  Summary ............................................................ 228
  References .......................................................... 230

Chapter 12  Sage Musings .............................................. 231
  Agility Gotta Be an Amalgamate .................................. 231
  Traditional Requirements-Gathering Techniques Are Passé .......... 233
  The MVP Paradigm Is Worth Considering .......................... 234
  Do Not Be a Prisoner of Events .................................. 235
  Predictive Analytics Is Not the Only Entry Point into Analytics .. 235
  Leadership Can Be an Acquired Trait .............................. 236
  Technology-Driven Architecture Is a Bad Idea ....................... 237
  Open Source Is Cool but to a Point ............................... 238
  Write Them Up However Trivial They May Seem ...................... 239
  Baseline Your Architecture on Core Strengths of Technology Products 240
  Summary ............................................................ 241
  References .......................................................... 241

Appendix A  25 Topic Goodies ....................................... 243
  What Is the Difference Between Architecture and Design? ........... 243
  What Is the Difference Between Architectural Patterns, Design Patterns, and a Framework? 243
  How Can We Compare a Top-Down Functional Decomposition Technique and an Object-Oriented Analysis and Design (OOAD) Technique? 244
  How Do Architecture Principles Provide Both Flexibility and Resilience to Systems Architecture? 245
  Why Could the Development of the Physical Operational Model (POM)
  Be Broken into Iterations? ........................................ 246
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is a Service-Oriented Architecture?</td>
<td>246</td>
</tr>
<tr>
<td>What Is an Event-Driven Architecture?</td>
<td>246</td>
</tr>
<tr>
<td>What Is a Process Architecture?</td>
<td>247</td>
</tr>
<tr>
<td>What Is a Technology Architecture?</td>
<td>248</td>
</tr>
<tr>
<td>What Is an Adapter?</td>
<td>248</td>
</tr>
<tr>
<td>What Is a Service Registry?</td>
<td>249</td>
</tr>
<tr>
<td>What Is a Network Switch Block?</td>
<td>249</td>
</tr>
<tr>
<td>What Are Operational Data Warehouses?</td>
<td>249</td>
</tr>
<tr>
<td>What Is the Difference Between Complex Event Processing (CEP) and</td>
<td></td>
</tr>
<tr>
<td>Stream Computing?</td>
<td>250</td>
</tr>
<tr>
<td>What Is the Difference Between Schema at Read and Schema at Write Techniques?</td>
<td>251</td>
</tr>
<tr>
<td>What Is a Triple Store?</td>
<td>251</td>
</tr>
<tr>
<td>What Is a Massively Parallel Processing (MPP) System?</td>
<td>252</td>
</tr>
<tr>
<td>What Is the Difference Between Supervised and Unsupervised Learning Techniques?</td>
<td>253</td>
</tr>
<tr>
<td>What Is the Difference Between Taxonomy and Ontology?</td>
<td>253</td>
</tr>
<tr>
<td>What Is Spark and How Does It Work?</td>
<td>254</td>
</tr>
<tr>
<td>What Are Some of the Advantages and Challenges of the Cloud Computing Platform and Paradigm?</td>
<td>256</td>
</tr>
<tr>
<td>What Are the Different Cloud Deployment Models?</td>
<td>257</td>
</tr>
<tr>
<td>What Is Docker Technology?</td>
<td>258</td>
</tr>
<tr>
<td>Summary</td>
<td>259</td>
</tr>
<tr>
<td>References</td>
<td>259</td>
</tr>
</tbody>
</table>

Appendix B  Elixir Functional Model (Continued)  ............... 261

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Level</td>
<td>261</td>
</tr>
<tr>
<td>Specified Level</td>
<td>264</td>
</tr>
<tr>
<td>Physical Level</td>
<td>267</td>
</tr>
</tbody>
</table>

Index .................................................. 269
Ah. Software architecture. A phrase that brings delight to some, grumblings to others, and apathy to far too many, particularly those who are far too busy slamming out code to bother with design.

And yet, as we know, all software-intensive systems have an architecture. Some are inten-tional, others are accidental, and far too many are hidden in the constellation of thousands upon thousands of small design decisions that accumulate from all that code-slamming.

Tilak takes us on a wonderful, approachable, and oh-so-very pragmatic journey through the ways and means of architecting complex systems that matter. With a narrative driven by a set of case studies—born from his experience as a practical architect in the real world—Tilak explains what architecture is, what it is not, and how it can be made a part of developing, delivering, and deploying software-intensive systems. I’ve read many books and papers about this subject—if you know me, you’ll know that I have a few Strong Opinions on the matter—but do know that I find Tilak’s approach based on a solid foundation and his presentation quite understandable and very actionable.

Architecting is not just a technical process, it’s also a human one, and Tilak groks that very important point. To that end, I celebrate how he interjects the hard lessons he’s learned in his career as a practical architect.

Architecture is important; a process of architecting that doesn’t get in the way but that does focus one on building the right system at the right time with the right resources is essential...and very practical.

Grady Booch
IBM Fellow and Chief Scientist for Software Engineering
Software architecture, as a discipline, has been around for half a century. The concept was introduced in the 1960s, drawing inspiration from the architecture of buildings, which involved developing blueprints that formulated designs and specifications of building architecture before any construction ever began. A blueprint of a building provides an engineering design of the functional aspects of the building—the floor space layout with schematics and measurements of each building artifact (for example, doors, windows, rooms, bathrooms, and staircases). The blueprint also provides detailed designs of the aspects needed to keep the building operational—the physics of the building foundation required to support the load of the building structure; the design of electrical cabling, water, and gas pipelines; and sewer systems needed for a fully operative and usable building.

True inspiration was drawn from the discipline of civil engineering (of building architectures) into information technology (IT); software architectures were broadly classified into functional architecture and operational architecture. The practice of software architecture started gaining momentum in the 1970s, and by the 1990s, it had become mainstream in the world of IT. At this time, architecture patterns were formulated. Patterns continue to evolve when recurrent themes of usage are observed; recurrences imply consistent and repeated application. Pattern development in software architecture presupposed that software architecture, as a discipline, was practiced enough to become mainstream and accepted as a formal discipline of study and practice.

With the complexity of IT Systems on the rise, IT projects have seen consistent and widespread use of software architectures. With more use comes diversity, or the emergence of various schools of thought that indoctrinate different views toward software architecture and popularize them through their adoption in the development of real-world software systems. With the growing number of variations and views toward software architectures, IT practitioners are typically
confused about which school of thought to adopt. As a case in point, have you found yourself asking some of the following questions?

- Because I have read so many books on architecture and have devoured so many journals and publications, how do I put the different schools of thought together?
- Which aspects of which schools of thought do I like more than others?
- Can the aspects complement each other?
- Where should I start when tasked with becoming an architect in a time-constrained, budget-constrained, complex software systems implementation?
- Can I succeed as a software architect?

I too have been in such a confused state. One of the toughest challenges for software architects is to find the best way to define and design a system’s or application’s software architecture. Capturing the essential tenets of any software architecture is as much a science as it is an art form. While the science lies in the proper analysis, understanding, and use of an appropriate description language to define the software architecture of the system, the art form assumes significance in defining a clear, crisp, nonredundant depiction used for effective communication with the different stakeholders of the system’s solution architecture. Software architects find it immensely challenging to determine how to capture the essential architecture artifacts in a way that clearly articulates the solution. While overengineering and excessive documentation add significant delays and associated risks to project delivery, a suboptimal treatment can result in the developer’s lack of comprehension regarding the solution architecture. Understanding the architecture is critical to adhere to the guidelines and constraints of technology and its use to design and develop the building blocks of the system. This gap can only widen with progression in the software development life cycle.

In 2008, I started writing a series of articles in the IBM developerWorks® journal; the focus was on documenting software architecture. I published four parts in the series and then for some personal reason could not continue. For the next few years, above and beyond the standard queries and accolades on the series topics, I started to receive a class of queries that got me increasingly thinking. Here are some snippets from these queries:

- “Dear Sir, I am using your article series as a part of my master’s thesis. May I know when your next set of articles is coming out?”
- “Mr. Mitra, We have embarked on an IT project in which we [have] adopted your architecture framework. Our project is stalled because the next article is not out. Please help.”

One fine morning it dawned on me that there must be a serious need for an end-to-end architecture treatment, one that is simple, crisp, comprehensible, prescriptive and, above all, practical enough to be executable. IT professionals and students of software engineering would significantly benefit from such a practical treatise on architecting software systems. It took me a while to finally put ink on paper; Practical Software Architecture: Moving from System Context
to Deployment represents all the collective wisdom, experience, learning, and knowledge in
the field of software architecture that I have gathered in more than 18 years of my professional
career. I have tried to write this book catering to a wide spectrum of readers, including

- Software architects, who will benefit from prescriptive guidance on a practical and
  repeatable recipe for developing software architectures.
- Project managers, who will gain an understanding and appreciation of the essential ele-
  ments required to develop a well-defined system architecture and account for just enough
  architecture activities in the project plan.
- Graduate students, who will find this book relevant in understanding how the theoretical
  premises of software architecture can actually be translated and realized in practice. This
  book is intended to be their long-time reference guide irrespective of technology
  advancements.
- Professors, who will use the book to help students transition from the theoretical aspects
  of software architecture to its real-world rendition, assisting students to become practical
  software architects.
- C-level and senior-level executives, who will benefit indirectly by gaining an awareness
  and appreciation for what it takes to develop well-formed system architectures for any IT
  initiative. This indirect knowledge may allow them to better appreciate IT architecture as
  a fundamental discipline in their company.

I intend this to be a practical how-to book with recipes to iteratively build any software
architecture through the various phases of its evolution. It shows how architectural artifacts may
be captured so that they are not only crisp, concise, precise, and well understood but also are just
enough in their practical application. Throughout the book, I have also used the terms “software,”
“systems,” and “solution” quite liberally and interchangeably to qualify the term architecture.
The liberal and interchangeable usage of the three terms is a conscious decision to work the mind
into such acceptance; they are used quite loosely in the industry.

On a philosophical note, the East and the West have been historically divided in their
acceptance of two forms of consciousness: the rational and the intuitive. Whereas the Western
world believes in and primarily practices rational, scientific, and deductive reasoning techniques,
the Eastern world places a premium on intuitive knowledge as the higher form in which aware-
ness (which is knowledge) is gained by watching (and looking inside one’s self; through self-
introspection) rather than gained only through experimental deductions. Being born and raised in
a culturally rich Bengali (in Kolkata, India) family, I firmly believe in the Eastern philosophies
of religion and knowledge, one in which conscious awareness is ultimately obtained through the
practice of conscious free will; the ultimate knowledge is gained through intuitive and inductive
reasoning. However, having been in the Western world for close to two decades, I do value
the scientific and rational knowledge form. I have come to believe that for us as mere mortals
to survive in this world of fierce competition, it is imperative that we master the rational and
scientifically derived knowledge, especially in the field of science, engineering, and IT. Once such a professional stability is attained, it is worthwhile, if not absolutely rewarding, to delve into the world of intuitive consciousness, of inductive reasoning—one through which we can attend moksha in life’s existentialism.

In this book, I have tried to share a prescriptive technique to help master practical ways of developing software architecture, through deductive and rational knowledge reasoning. My hope is that, if you can master the rational knowledge, you can turn your inner focus into the more mystical world of intuitive knowledge induction techniques. Solving the toughest architecture challenges is the Holy Grail; to be able to intuitively derive aspects of software architecture is the higher-level moksha we should all aim to achieve!

By the time you have finished reading this book and consuming its essence, I envision a newly emerged practical software architect in you. At least you will be a master of rational knowledge in this fascinating discipline of software architecture, paving the way into the world of mystical intuition, some of which I have only just started to experience!

P.S. If you are curious about the epigraphs at the start of each chapter, they were conjured up in the mind of yours truly!
I would first like to thank my wife, Taneea, and my mom, Manjusree, for giving me the time and inspiration to write this book. My uncle Abhijit has been the most persistent force behind me to make me believe that I could complete the book. And to my one and only son, Aaditya, for having consistently expressed his wonder regarding how his dad can write yet another book.

On the professional side, I convey my sincere gratitude to Ray Harishankar for supporting me in this gratifying authoring journey, right from its very inception; he is my executive champion. I would also like to thank my colleague Ravi Bansal for helping me review and refine the chapter on infrastructure; I relied on his subject matter expertise. My colleague from Germany, Bertus Eggen, devised a very nifty mathematical technique to help design the capacity model for network connectivity between servers, and I would like to thank Bert for giving me the permission to leverage his idea in my book. My sincere thanks go out to Robert Laird who has, so willingly, reviewed my book and given me such invaluable feedback. Many thanks to Craig Trim for sharing some of the inner details and techniques in natural language processing.

I would like to sincerely thank Grady Booch. I cannot be more humbled and honored to have Grady write the foreword for my book.

And to the Almighty, for giving us our son, Aaditya, born in 2010, who brings me unbridled joy; he is the one I look forward to in the years to come. He is already enamored with my “high-flying” professional lifestyle and wants to become like me; it will be my honest attempt in guiding him to set his bar of accomplishments much higher.
Tilak Mitra is a Chief Technology Officer at IBM, Global Business Services®. Tilak is an IBM Distinguished Engineer, with more than 18 years of industry experience in the field and discipline of IT, with a primary focus on complex systems design, enterprise architectures, applied analytics and optimization, and their collective application primarily in the field of industrial manufacturing, automation, and engineering, among various other adjacent industries. He is an influential technologist, strategist, well-regarded thought leader, and a highly sought-after individual to define and drive multidisciplinary innovations across IBM.

As the CTO, Tilak not only drives IBM’s technology strategy for the Strategic Solutions portfolio but also spearheads transformative changes in IBM’s top clients, developing innovative and new business models to foster their IT transformational programs.

Tilak is the co-author of two books—Executing SOA and SOA Governance—and has more than 25 journal publications. He is a passionate sportsperson, captains a champion cricket team in South Florida, and is also a former table tennis (ping pong) champion.

He currently lives in sunny South Florida with his wife and son. He can be reached at tilak_m@yahoo.com.

About the Author
This page intentionally left blank
This page intentionally left blank
Software Architecture: The *What* and *Why*

Unless I am convinced, I cannot put my heart and soul into it.

If you’re reading this chapter, I am going to assume that you are serious about following the cult of “The Practical Software Architect” and you would like to not only proudly wear the badge but also practice the discipline in your real-world software and systems development gigs and be wildly successful at it.

Software architects come in various flavors, and often they are interesting characters. Some architects work at a very high level engaging in drawing pictures on the back of a napkin or drawing a set of boxes and lines on a whiteboard, where no two boxes ever look the same. Others tend to get into fine-grained details too soon and often fail to see the forest for the trees; that is, see the bigger overarching architectural landscape. Still others are confused about what is the right mix. There is a need to level the playing field so that there is not only a common and comprehensible understanding of the discipline of software architecture, but also of what is expected of the role of the software architect, in order to be successful every time.

This chapter provides some background on the discipline of software architecture and some of the time-tested value drivers that justify its adoption. I end the chapter by laying some groundwork for the essential elements of the discipline that you and I, as flag bearers of the practical software architect cult, must formalize, practice, and preach.

How about a *The PSA* (pronounced “thepsa”) T-shirt?

Some Background

Software architecture, as a discipline, has been around for more than four decades, with its earliest works dating back to the 1970s. However, it is only under the pressures of increasing complexity hovering around the development of complex, mission-critical, and real-time systems that it has emerged as one of the fundamental constructs of mainstream systems engineering and software development.
Like any other enduring discipline, software architecture also had its initial challenges. However, this is not to say that it is free of all the challenges yet! Early efforts in representing the architectural constructs of a system were laden with confusing, inconsistent, imprecise, disorganized mechanisms that were used to diagrammatically and textually represent the structural and behavioral aspects of the system. What was needed was a consistent and well-understood pseudo- or metalanguage that could be used to unify all modes of representation and documentation of software architecture constructs and artifacts. The engineering and computer science communities, fostered by academic research, have made tremendous strides in developing best practices and guidelines around formalization of architecture constructs to foster effective communication of outcomes with the necessary stakeholders.

**The What**

Various research groups and individual contributors to the field of software engineering have interpreted software architecture, and each of them has a different viewpoint of how best to represent the architecture of a software system. Not one of these interpretations or viewpoints is wrong; rather, each has its own merits. The definition given by Bass, Clements, and Kazman (2012) captures the essential concept of what a software architecture should entail:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.

Now what does this definition imply?

The definition focuses on the fact that software architecture is comprised of coarse-grained constructs (a.k.a. software components) that can be considered building blocks of the architecture. Let’s call them architecture building blocks (ABB). Each such software component, or ABB (I use the terms interchangeably from here on), has certain externally visible properties that it announces to the rest of the ABBs. The internal details of how each software component is designed and implemented should not be of any concern to the rest of the system. Software components exist as black boxes—that is, internal details are not exposed—exposing only certain properties that they exhibit and that the rest of the software components can leverage to collectively realize the capabilities that the system is expected to deliver. Software architecture not only identifies the ABBs at the optimum level of granularity but also characterizes them according to the properties they exhibit and the set of capabilities they support. Capturing the essential tenets of the software architecture, which is defined by the ABBs and their properties and capabilities, is critical; therefore, it is essential to formalize the ways it is captured such that it makes it simple, clear, and easy to comprehend and communicate.

Architecture as it relates to software engineering is about decomposing or partitioning a single system into a set of parts that can be constructed modularly, iteratively, incrementally, and independently. These individual parts have, as mentioned previously, explicit relationships
between them that, when weaved or collated together, form the system—that is, the application’s software architecture.

Some confusion exists around the difference between architecture and design. As Bass, Clements, and Kazman (2012) pointed out, all architectures are designs, but not all designs are architectures. Some design patterns that foster flexibility, extensibility, and establishment of boundary conditions for the system to meet are often considered architectural in nature, and that is okay. More concretely, whereas architecture considers an ABB as a black box, design deals with the configuration, customization, and the internal workings of a software component—that is, the ABB. The architecture confines a software component to its external properties. Design is usually much more relaxed, since it has many more options regarding how to adhere to the external properties of the component; it considers various alternatives of how the internal details of the component may be implemented.

It is interesting to observe that software architecture can be used recursively, as illustrated in Figure 2.1.

![Figure 2.1 Illustrative example of recursive component dependencies.](image)

Referring to Figure 2.1, consider a software component (C₁ representing a Classroom) that is a part of a system’s software architecture. The software architect shares this software component (among others), along with its properties, functional and nonfunctional capabilities, and its relationships to other software components, to the system designer—the collection of ABBs along with their interrelationships and externally visible properties represents an *architecture blueprint*. The designer, after analyzing the software component (C₁), decides that it may be
broken down into some finer-grained components (C11 representing a Table object, C12 representing a Chair object, and C13 representing a Blackboard object), each of which provides some reusable functionality that would be used to implement the properties mandated for C1. The designer details C11, C12, C13, and their interfaces. The designer may consider C11, C12, and C13 as architectural constructs, with explicitly defined interfaces and relationships, for the software component C1. Then C11, C12, and C13 may need to be further elaborated and designed to address their internal implementations. Hence, architecture principles can be used recursively as follows: divide a large complex system into small constituent parts and then focus on each part for further elaboration.

Architecture, as mentioned previously, confines the system to using the ABBs that collectively meet the behavioral and quality goals. It is imperative that the architecture of any system under consideration needs to be well understood by its stakeholders: those who use it for downstream design and implementation and those who fund the architecture to be defined, maintained, and enhanced. And although this chapter looks more closely at this issue later on, it is important to highlight the importance of communication: architecture is a vehicle of communicating the IT System with the stakeholder community.

The Why

Unless I am convinced about the need, the importance, and the value of something, it is very difficult for me to motivate myself to put in my 100 percent. If you are like me and would like to believe in the value of software architecture, read on!

This section illustrates some of the reasons that convinced me of the importance of this discipline and led me to passionately and completely dedicate myself to practicing it.

A Communication Vehicle

Software architecture is the blueprint on which an IT System is designed, built, deployed, maintained, and managed. Many stakeholders expect and hence rely on a good understanding of the system architecture. However, one size does not fit all: a single view of the architecture would not suffice to satisfy the needs and expectations of the stakeholder community; multiple architecture viewpoints are needed.

Different views of the architecture are required to communicate its essence adequately to the stakeholders. For example, it is important to communicate with business sponsors in their own language (for example, a clear articulation of how the architecture addresses business needs). It should also communicate and assure the business stakeholders that it does not look like something that has been tried before and that has failed. The architecture representation should also illustrate how some of the high-level business use cases are realized by combining the capabilities of one or more ABBs. The representation (a.k.a., a viewpoint, which this chapter elaborates on later) and the illustrations should also focus on driving the value of the architecture blueprint.
as the foundation on which the entire system will be designed and built. The value drivers, in business terms, will ultimately need to ensure that there is adequate funding to maintain the vitality of the architecture until, at least, the system is deployed, operational, and in a steady state.

For the technical team, there should be multiple and different architecture representations depending on the technology domain. Following are a few examples:

- An application architect needs to understand the application architecture of the system that focuses on the functional components, their interfaces, and their dependencies—the functional architecture viewpoint.
- An infrastructure architect may be interested in (but not limited to) understanding the topology of the servers, the network connectivity between the servers, and the placement of functional components on servers—the operational architecture viewpoint.
- A business process owner would certainly be interested in understanding the various business processes that are enabled or automated by orchestrating the features and functions supported by the system. A business process is typically realized by orchestrating the capabilities of one or more business components. A static business component view, along with a dynamic business process view, would illustrate what business process owners may be interested in—the business architecture viewpoint.

Effective communication of the architecture drives healthy debates about the correct solution and approach; various alternatives and trade-offs may be analyzed and decisions made in concert. This not only ensures that the stakeholders are heard but also increases the quality of the architecture itself.

Communicating the architecture in ways that ensure various stakeholders’ understanding of its value and what is in it for them, while also having their active participation in its evolution, is key to ensuring that the vitality of the architecture is appropriately maintained.

**Influences Planning**

Recall the fact that any software architecture can be defined, at a high level, by a set of ABBs along with their interrelationships and dependencies. Recall also that an ABB can be deconstructed into a set of components that also exhibit interrelationships and dependencies. In a typical software development process, the functionalities of the system are usually prioritized based on quite a few parameters: urgency of feature availability and rollout, need to tackle the tough problems first (in software architecture parlance, these problems often are called architecturally significant use cases), quarterly capital expenditure budget, and so on. Whatever the reason may be, some element of feature prioritization is quite common.

Dependencies between the ABBs provide prescriptive guidance on how software components may be planned for implementation (see Figure 2.2).
Consider a scenario (as in Figure 2.2) in which components C2 and C3 depend on the availability of C1’s functionality, while C2 and C3 themselves are independent of each other. The architect can leverage this knowledge to influence the project planning process. For example, the architect may perform the design of C1, C2, and C3 in parallel if sufficient resources (designers) are available; however, he may implement C1 first and subsequently parallelize the implementation of C2 and C3 (assuming sufficient resources are available). Proper knowledge of the architecture and its constituents is critical to proper project planning; the architect is often the project manager’s best friend, especially during the project planning process.

Seeing the value the architect brings to the planning process, the planning team has often been found to be greedy for more involvement of the architect. The complexity of the architecture components influences how time and resources (their skill sets and expertise levels) are apportioned and allocated.

If the stakeholders do not have a thorough understanding of the architecture, subsequent phases—design, implementation, test planning, and deployment—will have significant challenges in any nontrivial system development.

### Addresses Nonfunctional Capabilities

Addressing the nonfunctional capabilities of a software system is a key responsibility of its architecture. It is often said, and rightfully so, that lack of commensurate focus on architecting any system to support its nonfunctional requirements (NFR) often brings about the system’s failure and breakdown.

Extensibility, scalability, maintainability, performance, and security are some of the key constituents of a system’s nonfunctional requirements. NFRs are unique in that they may not always be component entities in their own right; rather, they require special attention of one or more functional components of the architecture. As such, the architecture may influence and augment the properties of such functional components. Consider a use case that is expected to have a response time of no more than one second. The system’s architecture determines that three ABBs—C1, C2, and C3—collectively implement the use case. In such a scenario, the nature and complexity of the supported features of the components dictate how much time each component
may get to implement its portion of the responsibility: C₁ may get 300 milliseconds, C₂ may get 500 milliseconds, and C₃ may get 200 milliseconds. You may start finding some clues from here how ABBs get decorated with additional properties that they need to exhibit, support, and adhere to.

A well-designed and thought-out architecture assigns appropriate focus to address the key nonfunctional requirements of the system, not as an afterthought but during the architecture definition phase of a software development life cycle.

The risks of failure, from a technical standpoint, are significantly mitigated if the nonfunctional requirements are appropriately addressed and accounted for in the system architecture.

**Contracts for Design and Implementation**

One crucial aspect of software architecture is the establishment of best practices, guidelines, standards, and architecture patterns that are documented and communicated by the architect to the design and implementation teams.

Above and beyond communicating the ABBs, along with their interfaces and dependencies, the combination of best practices, guidelines, standards, and architecture patterns provides a set of constraints and boundary conditions within which the system design and implementation are expected to be defined and developed. Such constraints restrict the design and implementation team from being unnecessarily creative and channel their focus on adhering to the constraints rather than violating them.

As a part of the communication process, the architect ensures that the design and implementation teams recognize that any violation of the constraints breaks the architecture principles and contract of the system. In some special cases, violations may be treated and accepted as exceptions if a compelling rationale exists.

**Supports Impact Analysis**

Consider a situation, which presumably should not be too foreign to you, in which there is scope creep in the form of new requirements. The project manager needs to understand and assess the impact to the existing project timeline that may result from the new requirements.

In this situation, an experienced project manager almost inevitably reverts first and foremost to her lead architect and solicits help in exercising the required impact analysis.

Recall that any software architecture defines the ABBs and their relationships, dependencies, and interactions. The architect would perform some analysis of the new use case and determine the set of software components that would require modifications to collectively realize the new use case or cases. Changes to intercomponent dependencies (based on additional information or data exchange) are also identified. The impact to the project timeline thus becomes directly related to the number of components that require change, the extent of their changes, and also additional data or data sources required for implementation. The analyses can be further extended to influence or determine the cost of the changes and any risks that may be associated
with them. Component characteristics are a key metric to attribute the cost of its design, implementation, and subsequent maintenance and enhancements.

I cited five reasons to substantiate the importance of software architecture. However, I am certain that you can come up with more reasons to drive home the importance of architecture. I decided to stop here because I felt that the reasons cited here are good enough to assure me of its importance. And, staying true to the theme of this book, when I know that it is just enough, it is time to move on to the next important aspect. My objective, in this book, is to share my experiences on what is just enough, in various disciplines of software architecture, so that you have a baseline and frame of reference from which you can calibrate it to your needs.

**Architecture Views and Viewpoints**

Books, articles, research, and related publications on the different views of software architecture have been published. There are different schools of thought that prefer one architecture viewpoint over the other and, hence, practice and promote its adoption. In the spirit of this book’s theme, I do not devote a separate chapter to an exhaustive treatment of the different views of software architecture; rather, I present one that I have found to be practical and natural to follow and hence to use.

---

**Views and Viewpoints**

Philippe Kruchten (1995, November) was the pioneer who postulated the use of views and viewpoints to address the various concerns of any software architecture. Kruchten was a part of the IEEE 1471 standards body, which standardized the definitions of view and introduced the concept of a viewpoint, which, as published in his paper (see “References”), are as follows:

- **Viewpoint**—“A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.”
- **View**—“A representation of a whole system from the perspective of a related set of concerns.”

IBM (n.d.) defined a set of viewpoints called the IBM IT System Viewpoint Library. I have found it to be quite complete, with appropriate coverage of the various facets of a system’s architecture. The library consists of four basic viewpoints and six cross-cutting viewpoints. Figure 2.3 provides a pictorial representation.
The four basic viewpoints of the IBM IT System Viewpoint Library are the following:

- **Requirements**—Models elements that capture all the requirements placed on the system, including business, technical, functional, and nonfunctional requirements. Use cases and use case models are the most common means of capturing the requirements viewpoint.

- **Solution**—Models elements that define the solution satisfying the requirements and constraints; further organized into two categories:

  - **Functional**—Focuses on the model elements that are structural in nature and with which the system is built by not only implementing the elements but also wiring the relationships between the elements (both static and dynamic). The functional
architecture (the focus of Chapter 7, “The Functional Model”), broadly speaking, is the construct through which the details of this viewpoint are captured.

- **Operational**—Focuses on how the target system is built from the structural elements and how the functional view is deployed onto the IT environment (which consists of the network, hardware, compute power, servers, and so on). The operational model (the focus of Chapter 8, “The Operational Model”) is the most common architecture construct through which the details of this viewpoint are captured.

- **Validation**—Models elements that focus on assessing the ability of the system to deliver the intended functionality with the expected quality of service. Functional and nonfunctional test cases are often used as the validation criteria to attest to the system’s expected capabilities.

As shown in Figure 2.3, the four basic viewpoints are interrelated. The functional and operational viewpoints collectively realize (that is, implement and support) the requirements viewpoint; both the functional and operational viewpoints are validated for acceptance through the validation viewpoint. Note that the “solution” construct does not appear explicitly in Figure 2.3; for the sake of clarity, I have only shown the functional and operation constructs that collectively define the solution construct.

The library also contains six cross-cutting viewpoints, depicted in Figure 2.3 as concentric squares around the four basic viewpoints. The idea is to illustrate the point that the cross-cutting viewpoints influence one or more of the basic viewpoints.

The six cross-cutting viewpoints are as follows:

- **Application**—Focuses on meeting the system’s stated business requirements. The application architect plays the primary role in addressing this viewpoint.

- **Technical**—Focuses on the hardware, software, middleware (see Chapter 5, “The Architecture Overview,” for a definition), and packaged applications that collectively realize the application functionality and enable the application to run. The infrastructure and integration architects play the primary roles in addressing this viewpoint.

- **Systems Management**—Focuses on post-deployment management, maintenance, and operations of the system. The application maintenance and management teams play the primary roles in addressing this viewpoint.

- **Availability**—Focuses on addressing how the system will be made and kept available (for example, 99.5 percent uptime) per the agreed-upon service-level agreements. The infrastructure architect plays the primary role in addressing this viewpoint, with support from the application and the middleware architects.

- **Performance**—Focuses on addressing the performance of the system (for example, 400 milliseconds average latency between user request and the system response) per
the agreed-upon service-level agreements. The application architect plays the primary role in addressing this viewpoint, with support from the middleware and infrastructure architects.

- **Security**—Focuses on addressing the security requirements of the system (for example, single sign-on, security of data transfer protocol, intrusion avoidance, among others). Some of the security requirements—for example, single sign-on—are addressed primarily by the application architect role, whereas other requirements such as data protocols (HTTPS, secure sockets) and intrusion avoidance are addressed primarily by the infrastructure architects.

There are many more details behind each of the basic and cross-cutting viewpoints. Each viewpoint has a set of elements that collectively characterize and define their responsibilities. Understanding them can provide key insights into how each viewpoint may be realized. Although there are many details behind each of the basic and cross-cutting viewpoints, the idea here is to acknowledge their existence and realize the fact that any system’s overall architecture has to typically address most, if not all, of the viewpoints. Awareness is key!

After having personally studied a handful of viewpoint frameworks, I feel that most, if not all, of them have a degree of commonality in their fundamental form. The reason is that each of the frameworks sets about to accomplish the same task of establishing a set of complementary perspectives from which the software architecture may be viewed, with the goal of covering the various facets of the architecture.

The choice of adopting a viewpoint framework, at least from the ones that are also quite established, hardened, and enduring, depends on your level of belief that it addresses your needs and your degree of comfort in its usability and adoption.

**Summary**

As humans, we need to be convinced of the value of the work we are undertaking in order to put our mind and soul into it, to believe in its efficacy so that we can conjure up a passionate endeavor to make it successful.

In this chapter I shared my rationale for and belief in the value of a well-defined software architecture in relation to developing a successful software system. I defined a software architecture (that is, the *What*) while also emphasizing its value (that is, the *Why*).

The chapter also introduced the notion of architecture views and viewpoints and provided an overview of one viewpoint library that I tend to follow quite often.

The next chapter highlights the various facets of software architecture that are described in the rest of the book. The fun begins!
References


This page intentionally left blank
## Index

### A
- ABBs (architecture building blocks), 8, 20, 77
  - dependencies between, 11
  - of ARA
    - Analytics Solutions ABBs, 222
    - Cognitive Computing ABBs, 228
    - Consumers ABBs, 223
    - Data Acquisition and Access ABBs, 218-219
    - Data and Information Security ABBs, 224
    - Data Integration and Consolidation ABBs, 221
    - Data Repository ABBs, 219
    - Data Type ABBs, 217
    - Descriptive Analytics ABBs, 225
    - Metadata ABBs, 223-224
    - Models ABBs, 219
    - Operational Analytics ABBs, 227
    - Predictive Analytics ABBs, 225-226
    - Prescriptive Analytics ABBs, 226
  - access layer, 173
  - accuracy, 112
  - adapters, 158, 248
  - addressing nonfunctional capabilities, 12-13
  - aggregation, 164
  - agility, 231-233
  - infrastructure framework, 233
  - MVP paradigm, 234-235
  - analytics, 199
  - cognitive computing, 204
  - descriptive analytics, 202
  - governance, 212
  - need for, 200-201
  - operational analytics, 201-202
  - predictive analytics, 202, 235-236
  - prescriptive analytics, 203-204
  - semi-structured layer, 217
  - structured data, 217
  - unstructured data, 218
  - analytics architecture reference model
    - foundation, 205-206
    - systems of engagement, 206
    - systems of insight, 206
  - Analytics as a Service, 178
  - Analytics Solutions ABBs, 222
  - Analytics Solutions layer (ARA), 210
  - “Analytics: The Speed Advantage,” 200
  - API-level integration, 158-160
  - APIs, 158
<table>
<thead>
<tr>
<th>Application Architecture</th>
<th>Application HA</th>
<th>Application Servers, Capacity Planning</th>
<th>Application Viewpoint</th>
<th>Approaches to Systems Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>188</td>
<td>191</td>
<td>16</td>
<td>152</td>
</tr>
</tbody>
</table>

**ARA**

| ABBs | Analytics Solutions ABBs | Cognitive Computing ABBs | Consumers ABBs | Data Acquisition and Access ABBs | Data and Information Security ABBs | Data Integration and Consolidation ABBs | Data Repository ABBs | Data Type ABBs | Descriptive Analytics ABBs | Operational Analytics ABBs | Predictive Analytics ABBs | Prescriptive Analytics ABBs | Vertical Layers | Data Governance Layer | Metadata Layer | Architecturally Significant Use Cases | Architecture | Baselining on Core Strengths of Technology Products | Blueprints | Compliance Factors | Example of | Importance of | Artifacts | As-a-Service Models | Analytics as a Service | IaaS | PaaS | SaaS | Solution as a Service | Assigning Components to Layers |
|------|-------------------------|-------------------------|-----------------|-------------------------------|----------------------------------|-------------------------------------|---------------------|----------------|-------------------------|---------------------------|-----------------------------|-----------------------------|----------------|---------------------|------------------|----------------------|----------|-------------------|-----------------|-----------------|----------|------------------|---------------------|
associating data entities with subsystems, 90-92
attributes
of architecture decisions, 68-69
of NFRs, 112-113
attributes of leaders, 237
availability, 112
HA, 180
application HA, 188
database HA, 188
disk subsystem HA, 184
hardware HA, 181-182
operating system HA, 182
SPoF, 180
availability viewpoint, 16

back office zone, 142
banking example of IT System view, 53
Batch integration pattern, 162
best practices for software architecture, 13
Best West Manufacturers case study, 1-4
BI (business intelligence), 202
business architecture viewpoint, 11, 41
Business Context, comparing with System Context, 23
business operating models, 42
business processes, 11
Business Process layer (Layered view), 50
business process modeling, 29
business rules, 87
business use cases
identifying, 85
versus system use case, 4
capacity planning, 189, 192
application servers, 191
database servers, 191-192
web servers, 190
capturing
architecture decisions, 67-70
interface details, 89
System Context, 25
case study, 30-31, 36
information flows, 28-29
case studies
architecture decisions, 72-74
Best West Manufacturers, 1-4
Elixir
architecture overview, 57, 60-62
functional model, 99-103, 106, 261, 264-267
infrastructure, 192-194
Integration view, 166-170
OM, 141, 144-147
System Context, 30-31, 36
CBM (Component Business Modeling), 79-81
accountability levels, 80
business competencies, 79
CEP (complex event processing), 250
channels, 27
cloud computing, 256-257
As-a-Service models
IaaS, 177
PaaS, 178
SaaS, 178
deployment models, 257-258
hosting, 176
CMS, 178-180
hybrid cloud deployment models, 177
private cloud deployment models, 177
public cloud deployment models, 176
virtualization, 139
CMS (Cloud Management Services), 178-180
cognitive computing, 204, 216, 228
collaboration diagrams, 85
COM (conceptual operational model), 114
developing, 114
defining zones and locations, 115-116
identifying components, 116-117
placing the components, 118
DUs, linking, 122
for Elixir case study, 141, 146
rationalizing, 123-125
retail example, 114, 122
validating, 123-125
communicating best practices, 13
comparing
architecture and design, 9
Business Context and System Context, 23
compatibility, 112
completeness DLT, 67
complexity of integration, 152
compliance factors for architecture decisions, 66-67
component architecture, 20
component meta-model, 94
component responsibility matrix, 86
components
assignment to layers, 94-96
of COM
defining, 116-117
placing, 118
identifying, 83-84
interaction at specified design level, 92-94
interface details, capturing, 89
composite business services, 159
Composition Service topology, 160
conceptual architecture of the IT System, 40
conceptual-level design, 81
conceptual models, 245
contectual nodes, 121
connections, implementing in POM, 131-137
Consumers ABBs, 223
Consumers layer (ARA), 210
containers, Docker technology, 258-259
Core Business Processes
component artifacts, 44
core layer, 173
creating architecture decisions, 67-72
cross-cutting viewpoints, 16-17
CRUD, 92
custom enterprise models, 220

data, velocity, 201
Data Acquisition and Access ABBs, 218-219
Data Acquisition and Access layer (ARA), 208
Data and Information component artifacts, 45
Data and Information Security ABBs, 224
database HA, 188
database servers, capacity planning, 191-192
data centers, 176
data entities, associating with subsystems, 90-92
Data Governance layer (ARA)
analytics governance, 212
integration governance, 211
Data Information and Security layer (ARA), 212
data/information architecture, 41
Data Integration and Consolidation ABBs, 221
Data Integration and Consolidation layer (ARA), 209
data-level integration, 154-155
Data Repository ABBs, 219
Data Repository layer (ARA), 209
data type ABBs, 217
data types layer (ARA), 208
data virtualization, 221
DDUs (data deployable units), 118-119
decisions, architecture decisions, 20
DeepQA, 204, 252-253
defining
components of COM, 116-117
location of system components, 115-116
software architecture, 8
System Context, 23
delivery channels, 27
dependencies between ABBs, 11
deployment models, cloud computing, 257-258
Descriptive Analytics, 202, 213
Descriptive Analytics ABBs, 225
descriptive modeling, 225
design
best practices, 13
versus architecture, 9
developing
architecture decisions, 67-72
functional model, 81
associating data entities with subsystems, 90-92
component assignment to layers, 94-96
component interaction, 92-94
component responsibility matrix, 86
interface specification, 88-90
logical-level design, 82-85
physical-level design, 96-99
specified-level design, 85
OM
COM, 114-118, 123-125
POM, 131-141
SOM, 125-128, 131
technical services, 125
development of OM, 113
diagrams
architecture overview, 39
Business Context, 24
Enterprise view, 42-43
Core Business Processes component artifacts, 44
Data and Information component artifacts, 45
Elixir case study, 57-60
Technology Enablers component artifacts, 46
upgrading, 47
Users and Delivery Channels component artifacts, 44
IT System view, 52
banking example, 53
Elixir case study, 61-62
nodes, 54-55
nonfunctional characteristics, 55
Layered view, 47-52
Elixir case study, 60-61
vertical layers, 49
System Context
channels, 27
external systems, 27-28
for Elixir, 30
users, 26
dimensional analysis, 225
Direct Connection topology, 160
disadvantages of multitasking, 235
disk subsystem HA, 184
distribution layer, 173
DLPARs (dynamic LPARs), 182
DLTs (Decision Litmus Tests), 67
DMZ, 141
Docker technology, 258-259
documenting architecture decisions, 66
DR (disaster recovery), 189
DUs (deployable units), 118
DDUs, placing, 119
EDUs, placing, 120
linking, 122
PDUs, placing, 119-120
Dynamic Binding, 160
dynamic view of System Context, information flows, 28-29

E
EAI (Enterprise Application Integration), 160
EDA (event-driven architecture), 246-247
EDUs (execution deployable units), 118-120
EDWs, 249-250
Eggen, Bert, 135
elaboration, 109
Elixir case study
architecture decisions, 72-75
architecture overview
Enterprise view, 57-60
IT System view, 61-60
Layered view, 60-61
functional model case studies, 99-103, 106, 261-267
infrastructure, 192-194
Integration View case study, 166-170
operational model, 141, 144
COM, 141, 146
POM, 147
SOM, 146
System Context, developing, 30-31, 36
ensuring QoS, 138-139
enterprise data warehouse, 221
enterprise-level views, 20
enterprise mobile applications, 223
enterprise search, 223
Enterprise view, 40-43
Core Business Processes component artifacts, 44
Data and Information component artifacts, 45
Elixir case study, 57-60
Technology Enablers component artifacts, 46
upgrading, 47
Users and Delivery Channels component artifacts, 44
IT System view, 52
banking example, 53
Elixir case study, 61-62
nodes, 54-55
nonfunctional characteristics, 55
Layered view, 47-52
Elixir case study, 60-61
vertical layers, 49
System Context
channels, 27
external systems, 27-28
for Elixir, 30
users, 26
dimensional analysis, 225
EAI (Enterprise Application Integration), 160
EDA (event-driven architecture), 246-247
EDUs (execution deployable units), 118-120
EDWs, 249-250
Eggen, Bert, 135
elaboration, 109
Elixir case study
architecture decisions, 72-75
architecture overview
Enterprise view, 57-60
IT System view, 61-60
Layered view, 60-61
functional model case studies, 99-103, 106, 261-267
infrastructure, 192-194
Integration View case study, 166-170
operational model, 141, 144
COM, 141, 146
POM, 147
SOM, 146
System Context, developing, 30-31, 36
ensuring QoS, 138-139
enterprise data warehouse, 221
enterprise-level views, 20
enterprise mobile applications, 223
enterprise search, 223
Enterprise view, 40-43
Core Business Processes component artifacts, 44
Data and Information component artifacts, 45
Elixir case study, 57-60
Technology Enablers component artifacts, 46
upgrading, 47
Users and Delivery Channels component artifacts, 44
IT System view, 52
banking example, 53
Elixir case study, 61-62
nodes, 54-55
nonfunctional characteristics, 55
Layered view, 47-52
Elixir case study, 60-61
vertical layers, 49
System Context
channels, 27
external systems, 27-28
for Elixir, 30
users, 26
dimensional analysis, 225
EAI (Enterprise Application Integration), 160
EDA (event-driven architecture), 246-247
EDUs (execution deployable units), 118-120
EDWs, 249-250
Eggen, Bert, 135
elaboration, 109
Elixir case study
architecture decisions, 72-75
architecture overview
Enterprise view, 57-60
IT System view, 61-60
Layered view, 60-61
fault tolerance. See also capacity planning
  application HA, 188
database HA, 188
disk subsystem HA, 184, 187
hardware HA, 181-182
operating system HA, 182
RAID, 184, 187
SPoF, 181
federated data integration technique, 154
filters, 165
flexibility DLT, 67
functional architecture viewpoint, 11, 15
functional model, 20
developing, 81
Elixir functional model case study, 261-267
logical-level design, developing, 82-85
need for, 77
physical-level design, developing, 96-99
purpose of
  establishing traceability between architecture and design activities, 78
  establishing traceability between requirements and architecture, 79
  linking with operational model, 78
  managing system complexity, 78
  semantic levels, 81
  specified-level design, developing, 85-96
gathering requirements, 233-234
Governance layer (Layered view), 52
HA (High Availability), 180
  application HA, 188
database HA, 188
disk subsystem HA, 184
hardware HA, 181-182
operating system HA, 182
RAID, 184, 187
SPoF, 180
HADR (High Availability & Disaster Recovery), 188
horizontal layers, ARA, 208
  Analytics Solutions layer, 210
  Consumers layer, 210
  Data Acquisition and Access layer, 208
  Data Integration and Consolidation layer, 209
  Data Repository layer, 209
  Data Types layer, 208
  Models layer, 209
horizontal scalability, 138
hosting, 176
CMS, 178-180
hybrid cloud deployment models, 177
private cloud deployment models, 177
public cloud deployment models, 176
hybrid cloud deployment models, 177
IaaS (Infrastructure as a Service), 177
IBM IT System Viewpoint Library, cross-cutting viewpoints, 16-17
identifying
  business use cases, 85
  components, 83-84
data entities, 90
  specification nodes, 126
  subsystems, 82-83
technical components, 126-128, 131
IDUs (installation deployable units), 118
"-ilities," 111
impact analysis, 13
implementing nodes and connections in POM, 131-137
independence DLT, 67
industry standard models, 219
influences planning, 11-12
Information Architecture layer (Layered view), 51
information flows, 28-29
infrastructure, 21
  cloud computing
    As-a-Service models, 177-178
    CMS, 178
deployment models, 176
components, selecting, 131-134
Elixir Systems case study, 192-194
HA, 180
application HA, 188
database HA, 188
disk subsystem HA, 184
hardware HA, 181-182
operating system HA, 182
SPoF, 180
hosting, CMS, 178-180
network infrastructure model, 173-175
topologies, 174
yin and yang analogy, 172
infrastructure framework for agile development, 233
insight, 222
integration, 151
API-level integration, 158-160
approaches to, 152
complexity of, 152
data-level integration, 154
federated technique, 154
replication technique, 155
Elixir Integration View case study, 166-170
layers, 152
message-level integration, 156-158
service-level integration, 160
user interface integration, 153-154
integration governance, 211
Integration layer (Layered view), 51
integration patterns, 21, 161
Batch, 162
message routers, 165-166
message transformers, 166
pipes and filters, 165
Synchronous Batch Request-Response, 163
integrity DLT, 67
intercomponent dependencies, 12
interface details, capturing, 89
interface specification, 88-90
IT subsystems, 78
artifacts, 82
identifying, 82-83
IT System view, 20, 40, 52
banking example, 53
Elixir case study, 61-62
nodes, 54-55
nonfunctional characteristics, 55

J-K-L
KPIs, 227
Kruchten, Philippe, 14
Layered view 20, 40, 47, 52
ARA, 207
horizontal layers, 208-210
pillars, 207
vertical layers, 210-212
Elixir case study, 60-61
vertical layers, 49
layers
assigning components to, 94-96
integration, 152
leadership, 237
legacy adapters, 158
linking
DUs, 122
functional model with operational model, 78
location of system components, defining, 115-116
logical data model, 90
logical-level design, developing, 82
business use cases, identifying, 85
component identification, 83-84
subsystem identification, 82-83
LPAR (logical partitioning), 181
LXC (Linux Containers), 258

M
MAA (Maximum Availability Architecture), 188
maintainability, 112
managing system complexity, 78
matrix algebra, 134
message-level integration, 156-158
message routers, 165-166
message transformers, 166
metadata ABBs, 223-224
micro design, 98
mirroring, 184
Models ABBs, 219
Models layer (ARA), 209
modifiability, 112
MOM (message-oriented middleware), 156
MPLS/VPN (Multiprotocol Label Switching VPN), 175
MPP (massively parallel processing) systems, 252
multitasking, disadvantages of, 235
MVP (minimal valuable product), 234-235

N
network infrastructure model, 173-175
access layer, 173
core layer, 173
distribution layer, 173
networks. See also infrastructure cloud computing
As-a-Service models, 177-178
hosting, 176
hybrid cloud deployment models, 177
private cloud deployment models, 177
public cloud deployment models, 176-180
HA, 180
application HA, 188
database HA, 188
disk subsystem HA, 184
hardware HA, 181-182
operating system HA, 182
SPoF, 180
segmentation, 175
topologies, 133, 174
network switch blocks, 249
next best action, 222
NFRs (nonfunctional requirements), 12-13, 86
attributes, 112-113
HA, 180
nodes
implementing in POM, 131-137
IT System view, 54-55
nonfunctional characteristics, IT System view, 55

O
OM (operational model), 109
COM, 114
developing, 114-118, 123-125
Elixir case study, 141-146
retail example, 114, 122
development, 113
elaboration, 109
“-ilities,” 111
linking with functional model, 78
need for, 110
NFR attributes, 112
POM, 114
developing, 131-141
Elixir case study, 147
QoS, ensuring, 138-139
SOM, 114
developing, 125-128, 131
Elixir case study, 146
technical viability assessment, 128-129
traceability, 111
ontology, 220, 253-254
OOAD (object-oriented analysis and design), 244-245
open source technologies, 238-239
operating system HA, 182
Operational Analytics, 201-202, 215, 227
operational architecture, 11, 16
operational dashboard, 223
Operational layer (Layered view), 50

P
PaaS (Platform as a Service), 178
parallel development, 82
parity bits, 187
party checksum, 187
PDU (presentation deployable units), 118-120
performance, 112, 138-139
performance viewpoint, 16
physical-level design, developing, 96-99
physical models, 245
pillars of ARA, 207
Cognitive Computing, 216
Descriptive Analytics, 213
Operational Analytics, 215
Predictive Analytics, 214
Prescriptive Analytics, 214
pipes, 165
placing
components of COM, 118
DDUs, 119
EDUs, 120
PDU, 119-120
POM (physical operational model), 114, 246
  developing, 131-137
  for Elixir case study, 147
  nodes and connections, implementing, 131-137
  QoS, ensuring, 138-139
  rationalizing, 139-141
  validating, 139-141
  portability, 112
  “The Practical Software Architect,” 7
  precision, 112
  Predictive Analytics, 202, 214, 225-226, 235-236
  predictive asset optimization, 222
  predictive customer insight, 222
  Prescriptive Analytics, 203-204, 214, 226
  private cloud deployment models, 177
  problem solving, 239
  process architecture, 247-248
  process breakdown, 29
  public cloud deployment models, 177
  Publish-Subscribe, 164
  purpose of functional model establishing traceability between architecture and design activities, 78
  establishing traceability between requirements and architecture, 79
  linking with operational model, 78
  managing system complexity, 78
  QoS (quality of service), 138-139, 176
  QoS layer (Layered view), 51
  quality attributes, 111
  RAID 0, 184
  RAID 1, 184, 187
  RAID 5, 184
  RAID 6, 185
  RAID 10, 186
  rationalizing
    COM, 123-125
    POM, 139-141
    SOM, 128, 131
  real-time analytics, 201-202
  real-time model scoring, 227
  recommender systems, 222
  recursive use of software architecture, 9-10
  reliability, 67, 112
  replication data integration technique, 155
  reporting dashboard, 223
  reporting workbench, 225
  representing information flows, 28
  requirements gathering, 29, 233-234
  requirements viewpoint, 15
  retail example of COM, 114, 122
  road analogy for network topologies, 133
  roles of users, 27
  SaaS (Software as a Service), 178
  scalability, 113, 138
    horizontal scalability, 138
    vertical scalability, 138
  scale out, 138
  scale up, 138
  schema at read techniques, 251
  schema at write techniques, 251
  secured zone, 141
  security, 112
  security viewpoint, 17
  segmentation, 175
  selecting infrastructure components, 131-134
  semantic integration, 221
  semantic levels of functional model, 81
  semantic model, 155, 220
  semi-structured layer, 217
  send and forget processing model, 158
  Service Components layer (Layered view), 50
  service-level integration, 160
  service registries, 249
  Services layer (Layered view), 50
  SLAs, 173
  SOA (service-oriented architecture), 49, 246
  software architecture, 7-8
  ABBs, 8
  addressing nonfunctional capabilities, 12-13
  best practices, 13
Index

defining, 8
impact analysis, 13
influences planning, 11-12
recursive use of, 9-10
representations, 11
viewpoints, 10
  business architecture viewpoint, 11
  functional architecture viewpoint, 11
  operational architecture viewpoint, 11
Solution as a Service, 178
solution viewpoint, 15
solving problems, 239-240
SOM (specification operational model), 114
developing, 125
  identifying specification nodes, 126
  identifying technical components, 126-128
for Elixir case study, 146
rationalizing, 128-131
technical viability assessment, 128-129
validating, 128-131
Spark, 254-255
specification nodes, identifying, 126
specified-level design, developing, 85
  associating data entities with subsystems, 90-92
  component assignment to layers, 94-96
  component interaction, 92-94
  component responsibility matrix, 86
  interface specification, 88-90
specified models, 245
SPoF (single points of failure), 180
store and forward processing model, 158
stream computing, 250
striping, 184, 187
structured data, 217
subsystems, 78
  artifacts, 82
  associating with data entries, 90
  identifying, 82-83
supervised learning techniques, 253
Synchronous Batch Request-Response, 163
Synchronous Request-Response, 162
system complexity, managing, 78
system context, 20, 24
  capturing, 25
  case study, 30-31, 36
  defining, 23
  diagrams, 26
  channels, 27
  external systems, 27-28
dynamic view, information flows, 28-29
systems integration, 151
API-level integration, 158-160
approaches to, 152
complexity of, 152
data-level integration, 154
  federated technique, 154
  replication technique, 155
Elixir Integration View case study, 166-170
integration patterns, 161
  aggregation, 164
  Batch, 162
  message routers, 165-166
  message transformers, 166
  pipes and filters, 165
  Publish-Subscribe, 164
  Store and Forward, 164
  Synchronous Batch Request-Response, 163
  Synchronous Request-Response, 162
layers, 152
message-level integration, 156-158
service-level integration, 160
user interface integration, 153-154
systems management, 16, 113
systems of engagement, 206
systems of insight, 206
system use cases, 4, 85

T

tabular format for capturing architecture decisions, 69-70
taxonomies, 220, 253-254
technical architecture, 41
technical components, identifying, 126-128, 131
technical services, developing, 125
technical viability assessment of SOM, 128-129
technical viewpoint, 16
technology adapters, 29
technology agnostic views, 39
technology-driven architecture, 237-238, 248
Technology Enablers component artifacts, 46
ThePSA, 7
three-tier hierarchical network model, 173-175
TOGAF (The Open Group Architecture Framework), 41
Tonnage Per Hour, 213
top-down functional decomposition, 244
topologies, 174
traceability, 79
CBM, 79-81
accountability levels, 80
business competencies, 79
establishing
between architecture and design activities, 78
between requirements and architecture, 79
OM, 111
traits of leaders, 237
triple stores, 251
TSA (Tivoli System Automation), 188

U
UML (Unified Modeling Language), 83, 90
unstructured data, 218
unsupervised learning techniques, 253
untrusted zone, 141
upgrading Enterprise view, 47
usability, 112
use cases
architecturally significant use cases, 11
BWM case study, 2-4
business use cases, 4, 85
identifying, 85
system use cases, 85
user interface integration, 153-154
users
roles, 27
System Context diagram, 26
Users and Delivery Channels component artifacts, 44
validating
COM, 123-125
POM, 139-141
SOM, 128, 131
validation viewpoint, 16
validity DLT, 67
value creation, 200
velocity, 201
vertical layers
ARA, 210
Data Governance layer, 211-212
Data Information and Security layer, 212
in Layered view, 49
vertical scalability, 138, 191
viewpoints
cross-cutting viewpoints, 16-17
of software architecture, 10, 14
business architecture viewpoint, 11
functional architecture viewpoint, 11
operational architecture viewpoint, 11
views, 39
Enterprise view, 40-43
Core Business Processes component artifacts, 44
Data and Information component artifacts, 45
Elixir case study, 57, 60
Technology Enablers component artifacts, 46
upgrading, 47
Users and Delivery Channels component artifacts, 44
validating
COM, 123-125
POM, 139-141
SOM, 128, 131
validation viewpoint, 16
validity DLT, 67
value creation, 200
velocity, 201
vertical layers
ARA, 210
Data Governance layer, 211-212
Data Information and Security layer, 212
in Layered view, 49
vertical scalability, 138, 191
viewpoints
cross-cutting viewpoints, 16-17
of software architecture, 10, 14
business architecture viewpoint, 11
functional architecture viewpoint, 11
operational architecture viewpoint, 11
views, 39
Enterprise view, 40-43
Core Business Processes component artifacts, 44
Data and Information component artifacts, 45
Elixir case study, 57, 60
Technology Enablers component artifacts, 46
upgrading, 47
Users and Delivery Channels component artifacts, 44
IT System view, 40, 52
banking example, 53
Elixir case study, 61-62
nodes, 54-55
nonfunctional characteristics, 55
Layered view, 40, 47-52
Elixir case study, 60-61
vertical layers, 49
technology agnostic, 39
virtualization, cloud-based, 139
VLANs, 176
VPNs (virtual private networks), 175

W
Watson, 252-253
Web APIs, 160
web servers, capacity planning, 190
Web Services, 160
work products, 26
writing down your problems, 239

X-Y-Z
XOR logic, 187

yin and yang analogy of infrastructure, 172

zones, 115-116
  back office zone, 142
  DMZ, 141
  secured zone, 141
  untrusted zone, 1412