Practical Software Architecture

Moving from System Context to Deployment

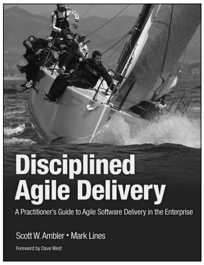
Tilak Mitra

Foreword by Grady Booch

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Related Books of Interest



Disciplined Agile Delivery

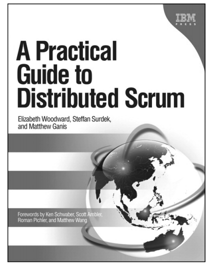
A Practitioner's Guide to Agile Software Delivery in the Enterprise

By Scott W. Ambler and Mark Lines

ISBN-13: 978-0-13-281013-5

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.



A Practical Guide to Distributed Scrum

By Elizabeth Woodward, Steffan Surdek, and Matthew Ganis

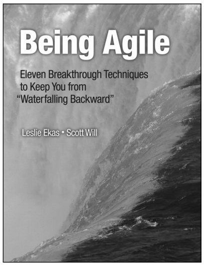
ISBN-13: 978-0-13-704113-8

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.

Sign up for the monthly IBM Press newsletter at
ibmpressbooks.com/newsletters

Related Books of Interest



Being Agile

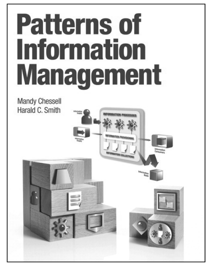
Eleven Breakthrough Techniques to Keep You from “Waterfalling Backward”

By Leslie Ekas, Scott Will

ISBN-13: 978-0-13-337562-6

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

ISBN-13: 978-0-13-315550-1

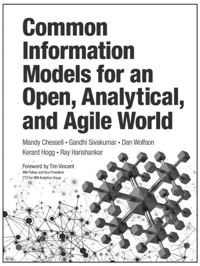
Use Best Practice Patterns to Understand and Architect Manageable, Efficient Information Supply Chains That Help You Leverage All Your Data and Knowledge

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.

IBM
Press™

Visit ibmpressbooks.com
for all product information

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

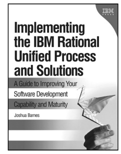
ISBN-13: 978-0-13-336615-0

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.

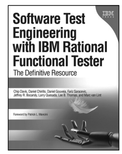


Implementing the IBM® Rational Unified Process® and Solutions

A Guide to Improving Your Software Development Capability and Maturity

Joshua Barnes

ISBN-13: 978-0-321-36945-1

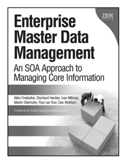


Software Test Engineering with IBM Rational Functional Tester

The Definitive Resource

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

ISBN-13: 978-0-13-700066-1

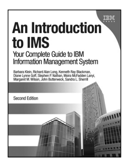


Enterprise Master Data Management

An SOA Approach to
Managing Core Information

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

ISBN-13: 978-0-13-236625-0

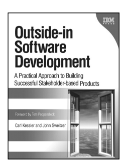


An Introduction to IMS

Your Complete Guide to IBM Information Management Systems, 2nd Edition

Barbara Klein, et al.

ISBN-13: 978-0-13-288687-1



Outside-in Software Development

A Practical Approach to Building Successful Stakeholder-based Products

Carl Kessler, John Sweitzer

ISBN-13: 978-0-13-157551-6

Sign up for the monthly IBM Press newsletter at
ibmpressbooks.com/newsletters

This page intentionally left blank

Practical Software Architecture

This page intentionally left blank

Practical Software Architecture

Moving from System Context to Deployment

Tilak Mitra

IBM Press
Pearson plc
New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ibmpressbooks.com

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

© Copyright 2016 by International Business Machines Corporation. All rights reserved.

Note to U.S. Government Users: Documentation related to restricted right. Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

IBM Press Program Managers: Steven M. Stansel, Ellice Uffer
Cover design: IBM Corporation
Editor-in-Chief: Dave Dusthimer
Marketing Manager: Stephane Nakib
Executive Editor: Mary Beth Ray
Publicist: Heather Fox
Editorial Assistant: Vanessa Evans
Managing Editor: Kristy Hart
Designer: Alan Clements
Senior Project Editor: Betsy Gratner
Copy Editor: Chuck Hutchinson
Indexer: Tim Wright
Compositor: Nonie Ratcliff
Proofreader: Debbie Williams
Manufacturing Buyer: Dan Uhrig

Published by Pearson plc
Publishing as IBM Press

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both: IBM, the IBM Press logo, developerWorks, Global Business Services, Maximo, IBM Watson, Aspera, Bluemix, z/OS, POWER5, DB2, Tivoli, WebSphere, and IBM PureData. SoftLayer is a registered trademark of SoftLayer, Inc., an IBM Company. A current list of IBM trademarks is available on the Web at "copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

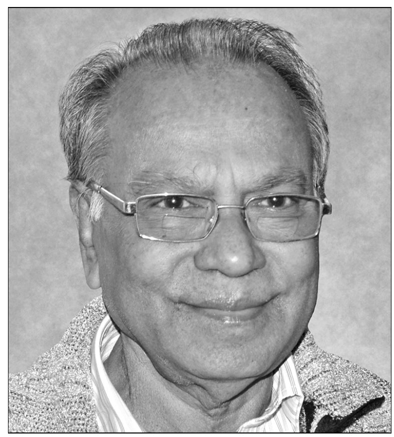
Library of Congress Control Number: 2015947371

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-376303-4
ISBN-10: 0-13-376303-X

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana. First printing: December 2015

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

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 intentional, 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

Preface

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 elements 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 awareness (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*!

Acknowledgments

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.

About the Author

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.

This page intentionally left blank

This page intentionally left blank

CHAPTER 2

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 T*he* 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.

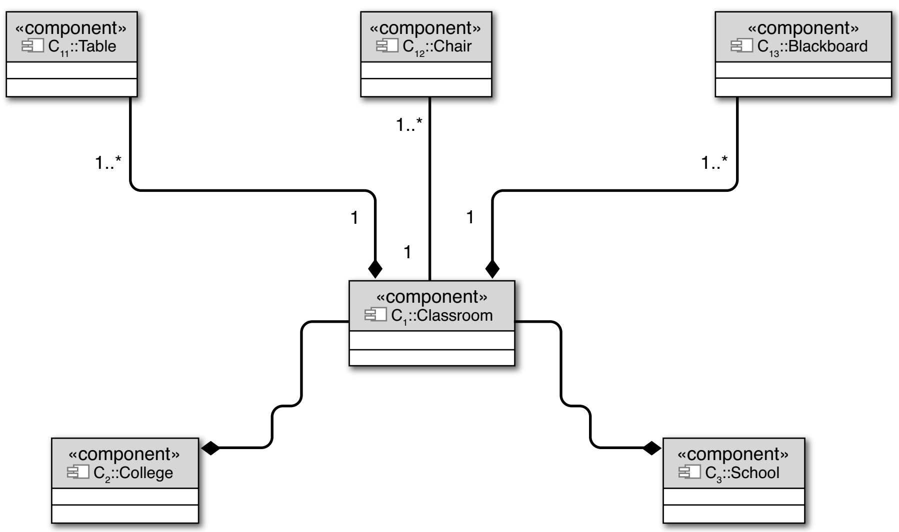


Figure 2.1 Illustrative example of recursive component dependencies.

Referring to Figure 2.1, consider a software component (C_1 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 (C_1), decides that it may be broken down into some finer-grained components (C_{11} representing a Table object, C_{12} representing a Chair object, and C_{13} representing a Blackboard object), each of which provides some reusable functionality that would be used to implement the properties mandated for C_1. The designer details C_{11}, C_{12}, C_{13}, and their interfaces. The designer may consider C_{11}, C_{12}, and C_{13} as architectural constructs, with explicitly defined interfaces and relationships, for the software component C_1. Then C_{11}, C_{12}, and C_{13} 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).

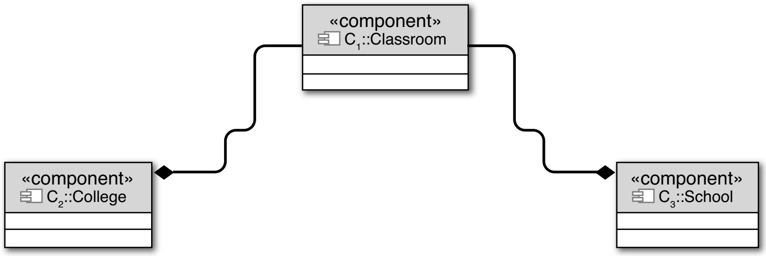


Figure 2.2 Illustrative example of intercomponent dependencies.

Consider a scenario (as in Figure 2.2) in which components C_2 and C_3 depend on the availability of C_1's functionality, while C_2 and C_3 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 C_1, C_2 and C_3 in parallel if sufficient resources (designers) are available; however, he may implement C_1 first and subsequently parallelize the implementation of C_2 and C_3 (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—C_1, C_2, and C_3—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: C_1 may get 300 milliseconds, C_2 may get 500 milliseconds, and C_3 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.

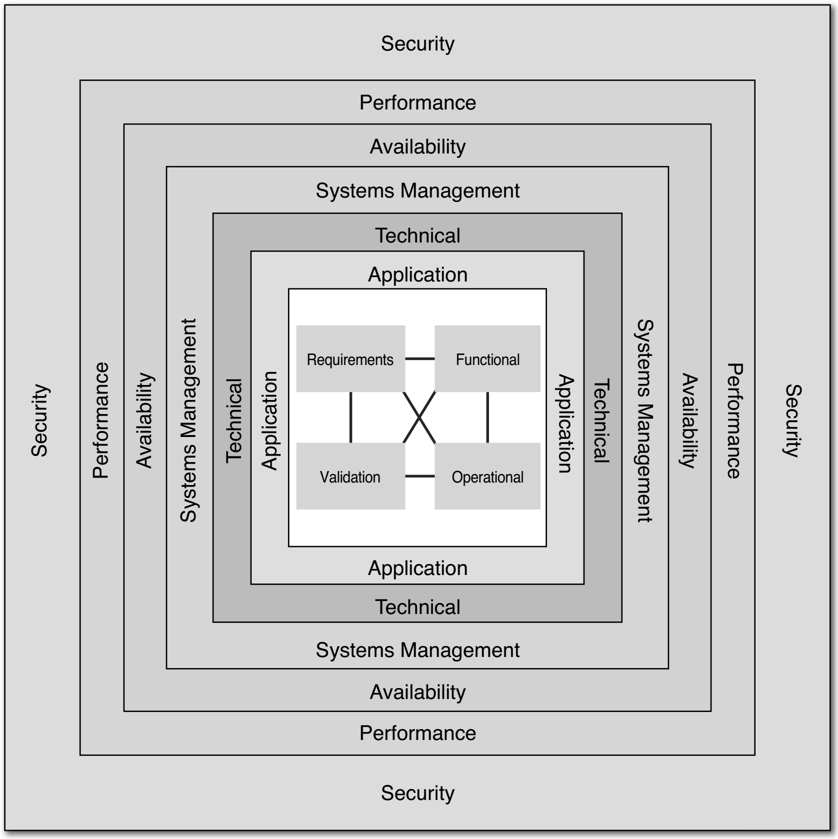


Figure 2.3 Viewpoints in the IBM IT System Viewpoint Library (see "References").

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

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice*, 3rd ed. (Upper Saddle River, NJ: Addison-Wesley Professional).

IBM. (n.d.) Introduction to IBM IT system viewpoint. Retrieved from http://www.ibm.com/developerworks/rational/library/08/0108_cooks-cripps-spaas/.

Kruchten, P. (1995, November). Architectural blueprints—The “4+1” view model of software architecture. *IEEE Software, 12*(6), 42–50. Retrieved from http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.

This page intentionally left blank

Index

A

D

E

F

G-H

I

J-K-L

M

T

U

V

W

X-Y-Z