·­·­·- 0

Software Architecture in Practice

Third Edition

Len Bass Paul Clements RickKazman

T• .. Addison-Wesley

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

A a

A a

·­·­·- 0

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

CMM, CMMI, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ATAM; Architecture Tradeoff Analysis Method; CMM Integration; COTS Usage-Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Appraiser; SCAMPI Lead Assessor; SCE; SEI; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

Special permission to reproduce portions of works copyright by Carnegie Mellon University, as listed on page 588, is granted by the Software Engineering Institute.

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

The authors 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.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales (800) 382-3419 corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales international@pearson. com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Software Engineering Institute I CarnegieMellon

The SEI Series in Software Engineering

A a

·­·­·- 0

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, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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

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

Bass, Len.

Software architecture in practice I Len Bass, Paul Clements, Rick Kazman. 3rd ed. p. em. (SEI series in software engineering) Includes bibliographical references and index. ISBN 978-0-321-81573-6 (hardcover : all<. paper) 1. Software architecture. 2. System design. I. Clements, Paul, 1955- II. Kazman, Rick. III. Title.

QA76.754.B37 2012 005.1 dc23

Copyright © 2013 Pearson Education, Inc.

2012023744

0

·­·­·-

A a

A a

on M.anagemcnt

�·�Addison-Wesley

The SEI Series in Software Engineering

Software Engineering Institute I CamegieMellon

Rdle-=tions

Wuu S. llu.til' 'rc:' ..... � .. "'--

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

The undertaking SEI Series of in the Software Carnegie Engineering Mellon Software represents Engineering is a collaborative lnstrtute (SEI) and

Addison-Wesley to develop and publish books on software engineering and

related topics. The common goal of the SEI and Addison-Wesley is to provide

the most current information on these topics in a form that is easily usable by

practitioners and students.

Books in the series describe frameworks, tools, methods, and technologies

designed to help organizations, teams, and individuals improve their technical

or management capabilities. Some books describe processes and practices for

developing higher-quality software, acquiring programs for complex systems, or

delivering services more effectively. Other books focus on software and system

architecture and product·line development. Still others, from the SEI's CERT

Program, describe technologies and practices needed to manage software

and network security risk. These and all books in the series address critical

problems in software engineering for which practical solutions are available.

PEARSON --

··•Addison-Wesley Cisco Press EXAM/CRAM � our: H ::;'.!,E�TICE sAM.s 1 Safarr> --

·­·­·- 0

Contents Preface v

Reader's Guide

Acknowledgments

Part One Introduction

Chapter 1 What Is Software Architecture?

1.1 What Software Architecture Is and What It Isn't

1.2 Architectural Structures and Views

1.3 Architectural Patterns

1.4 What Makes a "Good" Architecture?

1.5 Sutnmary

1. 6 For Further Reading

1. 7 Discussion Questions

Chapter 2 Why Is Software Architecture Important?

2.1 Inhibiting or Enabling a System's Quality Attributes

2.2 Reasoning About and Managing Change

2.3 Predicting System Qualities

2.4 Enhancing Communication among Stakeholders

2.5 Carrying Early Design Decisions

2.6 Defining Constraints on an Implementation

2.7 Influencing the Organizational Structure

2.8 Enabling Evolutionary Prototyping

2.9 Improving Cost and Schedule Estimates

2.10 Supng a Transferable, Reusable Model

2.1 1 Allowing Incorporation of Independently Developed Components

2.12 Restricting the Vocabulary of Design Alternatives

2.13 Providing a Basis for Training

2.14 Summary

2.15 For Further Reading

2.16 Discussion Questions

A a

Chapter 3 The Many Contexts of Software Architecture

311Architecture in a Technical Context 312 Architecture in a Project Life-Cycle Context 3.3 Architecture in a Business Context 3.4 Architecture in a ProfessionalContext

3 I 5 Stakeholders 316 How Is Architecture Influenced? 3�7 What Do Architectures Influence?

3.8 Summary 3.9 For Further Reading 3.10 Discusion Questions

Part Two Quality Attributes

Chapter 4 Understanding Quality Attributes

4.1 Architecture and Requirements

4.2 Functionality 4.3 Quality Attribute Considerions

4.4 Specifying Quality Attribute Requiremts

4.5 Achieving Quality Attributes through Tactics 4.6 Guiding Quality Design Decisions

4.7 Summary 4 I 8 For Further Reading

4.9 Discusion Questions

Chapter 5 A vail ability

5 I 1 Availability General Scenario 5.2 Tactics for Availability 5. 3 A Design Checklist for Availability

5.4 Summary

515 For Further Reading 5.6 Discussion Questions

Chapter 6 lnteroperability

611Interoperability General Scenario

6.2 Tactics for Interoperability

·­·­·- 0 A a

6.3 A Design Checklist for Interoperability 6.4 Summary

6.5 For Further Reading

6.6DiscussionQuestions

Chapter 7 Modifiability

7.1 Modifiability General Scenario 7.2 Tactics for Modifiability 7.3 A Design Checklist for Modifiability 7.4 Summary

7. 5 For Further Reading

7.6 Discusion Questions

Chapter 8 Performance

8.1 Performance General Scenario 8.2 Tactics for Performance 8.3 A Design Checklist for Performance 8.4 Summary

8.5 For Further Reading 8.6 DiscussionQuestions

Chapter 9 Security

9.1 Security General Scenario 9.2 Tactics for Security 9.3 A Design Checklist for Security 9.4 Summary 9.5 For Further Reading

9.6 Discusion Questions

Chapter 10 Testability

10.1 Testability General Scenario

10.2 Tactics for Testability 10.3 A Design Checklist for Testability 1 0.4 Summary

10.5 For Further Reading

10.6 Discusion Questions

·­·­·- 0 A a

0

·­·­·-

A a

A a

12.5 Dealing with "X-ability": Bringing a New Quality Attribute into the Fold

12.6 For Further Reading

12.7 Discussion Questions

Chapter 13 Architectural Tactics and Patterns

13.1 Architectural Patterns

13.2 Overview of the Patterns Catalog

13.3 Relationships between Tactics and Patterns

13.4 Using Tactics Together

13.5 Summary

13.6 For Further Reading

13.7 Discussion Questions

Chapter 14 Quality Attribute Modeling and Analysis

14.1 Modeling Architectures to Enable Quality Attribute Analysis

14.2 Quality Attribute Checklists

14.3 Thought Experiments and Back-of-the-Envelope Analysis

14.4 Experiments, Simulations, and Prototypes

14.5 Analysis at Different Stages of the Life Cycle

14.6 Summary

14.7 For Further Reading

14.8 Discussion Questions

Chapter 11 Usability

1 1.1 Usability General Scenario

1 1.2 Tactics for Usability

1 1 .3 A Design Checklist for Usability

1 1.4 Summary

1 1.5 For Further Reading

1 1.6 Discussion Questions

Chapter 12 Other Quality Attributes

12.1 Other Important Quality Attributes

12.2 Other Categories of Quality Attributes

12.3 Software Quality Attributes and System Quality Attributes

12.4 Using Standard Lists of Quality Attributes or Not

Part Three Architecture in the Life Cycle

Chapter 15 Architecture in Agile Projects

15.1 How Much Architecture?

15.2 Agility and Architecture Methods

15.3 A Brief Example of Agile Architecting

15.4 Guidelines for the Agile Architect 15.5 Summary 15.6 For Further Reading 15.7 DiscussionQuestions

Chapter 16 Architecture and Requirements

16. 1 Gathering ASRs from RequirementsDocuments

16.2 Gathering ASRs by Interviewing Stakeholders

·­·­·-

16.3 Gathering ASRs by Understanding the BusinessGoals

16.4Capturing ASRs in a Utility Tree 16.5 Tying the Methods Together

16.6 Summary 16.7 For Further Reading

16.8 Discusion Questions

Chapter17Designing an Architecture

17.1 Design Strategy 17.2 The Attribute-Driven Design Method 17.3 The Steps of ADD

17.4 Summary

17.5 For Further Reading 17.6 Discusion Questions

Chapter 18 Documenting Software Architectures

18. 1 Uses and Audiences for Architecture Documentation

18.2 Notations for Architecture Documentation

18.3Views

18.4 Choosing the Views 18.5 Combining Views

18.6 Building the Documentation Package

0 A a

·­·­·- 0 A a

18.7 Documenting Behavior

18.8 Architecture Documentation and Quality Attributes

18.9 Documenting Architectures That Change Faster Than You Can Document Them

18.10 Documenting Architecture in an Agile Development Project

18.11 Sutnmary

18.12 For Further Reading

18.13 Discussion Questions

Chapter 19 Architecture, Implementation, and Testing

19.1 Architecture and Implementation

19.2 Architecture and Testing

19.3 Summary

19.4 For Further Reading

19.5 Discussion Questions

Chapter 20 Architecture Reconstruction and Conformance

20.1 Architecture Reconstruction Process

20.2 Raw View Extraction

20.3 Database Construction

20.4 View Fusion

20.5 Architecture Analysis: Finding Violations

20.6 Guidelines

20.7 Summary

20.8 For Further Reading

20.9 Discussion Questions

Chapter 21 Architecture Evaluation

21.1 Evaluation Factors

21.2 The Architecture Tradeoff Analysis Method

21.3 Lightweight Architecture Evaluation

21.4 Summary

21.5 For Further Reading

21.6 Discussion Questions

Chapter 22 Management and Governance

22.1 Planning

·­·­·- 0

22.2 Organizing

22.3 Implementing

22.4 Measuring

22.5 Governance

22.6 Sutnmary

22.7 For Further Reading

22.8 Discussion Questions

Part Four Architecture and Business

Chapter 23 Economic Analysis of Architectures

23.1 Decision-Making Context

23.2 The Basis for the Economic Analyses

23.3 Putting Theory into Practice: The CBAM

23.4 Case Study: The NASA ECS Project

23.5 Summary

23.6 For Further Reading

23.7 Discussion Questions

Chapter 24 Architecture Competence

24.1 Competence of Individuals: Duties, Skills, and Knowledge of Architects

24.2 Competence of a Software Architecture Organization

24.3 Summary

24.4 For Further Reading

24.5 Discussion Questions

Chapter 25 Architecture and Software Product Lines

25.1 An Example of Product Line Variability

25.2 What Makes a Software Product Line Work?

25.3 Product Line Scope

25.4 The Quality Attribute of Variability

25.5 The Role of a Product Line Architecture

25.6 Variation Mechanisms

25.7 Evaluating a Product Line Architecture

25.8 Key Software Product Line Issues

25.9 Summary

A a

25.10 For Further Reading

2511 Discussion Questions

Part Five The Brave New World

Chapter 26 Architecture in the Cloud

261 1 Basic Cloud Definitions 2612 Service Models and Deployment Options

26.3 Economic Justification 26.4 BaseMechanisms

2615 Sample Technologies

26�6 Architecting in a Cloud Environment

2617 Summary

2618 For Further Reading

2619 DiscussionQuestions

Chapter27 Architectures for the Edge

2711 The Ecosystem of Edge-Dominant Systems 27�2 Changes to the Software DevelopmentLife Cycle

2713 Implications for Architecture

27.4 Implications of the Metropolis Model

27 I 5 Summary 27 I 6 For Further Reading 27�7 Discussion Questions

Chapter28 EpilogueReferences

About the Authors

Index

·­·­·- 0 A a

A a

Preface

·­·­·- 0

-Benjamin Franklin

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

have on the life cycle, organizations, and management.

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

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

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

We have thoroughly reworked many of the topics covered in this edition. In particular, we realize that the methods we present for architecture design, analysis, and documentation are one version of how to achieve a particular goal, but there are others. This led us to separate the methods that we present in detail from their underlying theory. We now present the theory first with specific methods given as illustrations of possible realizations of the theories. The new topics in this edition include architecture-centric project management; architecture competence; requirements modeling and analysis; Agile methods; implementation and testing; the cloud; and the edge.

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

I should have no objection to go over the same life fi"om its beginning to the end: requesting only the advantage authors have, of correcting in a [third} edition the faults ofthefirst [two}.

·­·­·- 0 A a

Reader's Guide

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

• Technical. What technical role does the software architecture play in the system or systems of which it's a part?

• Project. How does a software architecture relate to the other phases of a software development life cycle?

• Business. How does the presence of a software architecture affect an organization's business environment?

• Professional. What is the role of a software architect in an organization or a development project?

Part Two is focused on technical background. Part Two describes how decisions are made. Decisions are based on the desired quality attributes for a system, and Chapters 5-1 1 describe seven different quality attributes and the techniques used to achieve them. The seven are availability, interoperability, maintainability, performance, security, testability, and usability. Chapter 12 tells you how to add other quality attributes to our seven, Chapter 13 discusses patterns and tactics, and Chapter 14 discusses the various types of modeling and analysis that are possible.

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

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

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

·­·­·- 0 A a

Acknowledgments

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

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

The issue of build efficiency in Chapter 12 came from Rolf Siegers and John McDonald of Raytheon. John Klein and Eltjo Poort contributed the "abstract system clock" and "sandbox mode" tactics, respectively, for testability. The list of stakeholders in Chapter 3 is from Documenting Software Architectures: Views and Beyond, Second Edition. Some of the material in Chapter 28 was inspired by a talk given by Anthony Lattanze called "Organizational Design Thinking" in 201 1 .

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

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

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

0 A a

·­·­·-

What is a software architecture? What is it good for? How does it come to be? What effect does its existence have? These are the questions we answer in Part I.

Chapter 1 deals with a technical perspective on software architecture. We define it and relate it to system and enterprise architectures. We discuss how the architecture can be represented in different views to emphasize different perspectives on the architecture. We define patterns and discuss what makes a "good" architecture.

In Chapter 2, we discuss the uses of an architecture. You may be surprised that we can find so many -ranging from a vehicle for communication among stakeholders to a blueprint for implementation, to the carrier of the system's quality attributes. We also discuss how the architecture provides a reasoned basis for schedules and how it provides the foundation for training new members on a team.

Finally, in Chapter 3, we discuss the various contexts in which a software architecture exists. It exists in a technical context, in a project life-cycle context, in a business context, and in a professional context. Each of these contexts defines a role for the software architecture to play, or an influence on it. These impacts and influences define the Architecture Influence Cycle.

Part One. Introduction

A a

·­·­·- 0

Writing (on our part) and reading (on your part) a book about software architecture, which distills the experience of many people, presupposes that

1. having a software architecture is important to the successful development of a software

system and

2. there is a sufficient, and sufficiently generalizable, body of knowledge about software

architecture to fill up a book.

One purpose of this book is to convince you that both of these assumptions are true, and once you are convinced, give you a basic knowledge so that you can apply it yourself.

Software systetns are constructed to satisfy organizations' business goals. The architecture is a bridge between those (often abstract) business goals and the final (concrete) resulting system. While the path from abstract goals to concrete systems can be complex, the good news is that software architectures can be designed, analyzed, documented, and implemented using known techniques that will support the achievement of these business and mission goals. The complexity can be tamed, made tractable.

These, then, are the topics for this book: the design, analysis, documentation, and implementation of architectures. We will also examine the influences, principally in the form of business goals and quality attributes, which inform these activities.

In this chapter we will focus on architecture strictly from a software engineering point of view. That is, we will explore the value that a software architecture brings to a development project. (Later chapters will take a business and organizational perspective.)

1. What Is Software Architecture?

Good judgment is usually the result of experience. And experience is frequently the result of bad judgment. But to learn from the experience of

others requires those who have the experience to share the knowledge with those who follow.

-Barry LePatner

·­·­·- 0 A a

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

There are many definitions of software architecture, easily discoverable with a web search, but the one we like is this one:

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

This definition stands in contrast to other definitions that talk about the system's "early" or "major" design decisions. While it is true that many architectural decisions are made early, not all are:­ especially in Agile or spiral-development projects. It's also true that very many decisions are made early that are not architectural. Also, it's hard to look at a decision and tell whether or not it's "major." Sometimes only time will tell. And since writing down an architecture is one of the architect's most important obligations, we need to know now which decisions an architecture comprises.

Structures, on the other hand, are fairly easy to identify in software, and they form a powerful tool for system design.

Let us look at some of the implications of our definition.

Architecture Is a Set of Software Structures

This is the first and most obvious implication of our definition. A structure is simply a set of elements held together by a relation. Software systems are composed of many structures, and no single structure holds claim to being the architecture. There are three categories of architectural structures, which will play an important role in the design, documentation, and analysis of architectures:

1. First, some structures partition systems into implementation units, which in this book we call modules. Modules are assigned specific computational responsibilities, and are the basis of work assignments for progratruning teams (Team A works on the database, Team B works on the business rules, Team C works on the user interface, etc.). In large projects, these elements (modules) are subdivided for assignment to subteams. For example, the database for a large enterprise resource planning (ERP) implementation might be so complex that its implementation is split into many parts. The structure that captures that decomposition is a kind of module structure, the module decomposition structure in fact. Another kind of module structure emerges as an output of object-oriented analysis and design class diagratns. If you aggregate your modules into layers, you've created another (and very useful) module structure. Module structures are static structures, in that they focus on the way the system's functionality is divided up and assigned to implementation teams.

2. Other structures are dynamic, meaning that they focus on the way the elements interact with

each other at runtime to carry out the system's functions. Suppose the system is to be built as a set of services. The services, the infrastructure they interact with, and the synchronization and interaction relations among them form another kind of structure often used to describe a system. These services are made up of (compiled from) the programs in the various implementation units that we just described. In this book we will call runtime structures component-and-connector (C&C) structures. The term component is overloaded in software

0 ·­·­·-

A a

engineering. In our use, a component is always a runtime entity.

3. A third kind of structure describes the mapping from software structures to the system's

organizational, developmental, installation, and execution environments. For example, modules are assigned to teams to develop, and assigned to places in a file structure for implementation, integration, and testing. Components are deployed onto hardware in order to execute. These mappings are called allocation structures.

Although software comprises an endless supply of structures, not all of them are architectural. For example, the set of lines of source code that contain the letter "z," ordered by increasing length from shortest to longest, is a software structure. But it's not a very interesting one, nor is it architectural. A structure is architectural if it supports reasoning about the system and the system's properties. The reasoning should be about an attribute of the system that is important to some stakeholder. These include functionality achieved by the system, the availability of the system in the face of faults, the difficulty of making specific changes to the system, the responsiveness of the system to user requests, and many others. We will spend a great deal of time in this book on the relationship between architecture and quality attributes like these.

Thus, the set of architectural structures is not fixed or limited. What is architectural is what is useful in your context for your system.

Architecture Is an Abstraction

Because architecture consists of structures and structures consist of elementsl and relations, it follows that an architecture comprises software elements and how the elements relate to each other. This means that architecture specifically omits certain information about elements that is not useful for reasoning about the system in particular, it omits information that has no ramifications outside of a single element. Thus, an architecture is foremost an abstraction of a system that selects certain details and suppresses others. In all modem systems, elements interact with each other by means of interfaces that partition details about an element into public and private parts. Architecture is concerned with the public side of this division; private details of elements details having to do solely with internal implementation are not architectural. Beyond just interfaces, though, the architectural abstraction lets us look at the system in terms of its elements, how they are arranged, how they interact, how they are composed, what their properties are that support our system reasoning, and so forth. This abstraction is essential to taming the complexity of a system we simply cannot, and do not want to, deal with all of the complexity all of the time.

1. In this book we use the term "element" when we mean either a module or a component, and don't want to

distinguish.

Every Software System Has a Software Architecture

Every system can be shown to comprise elements and relations among them to support some type of reasoning. In the most trivial case, a system is itself a single element an uninteresting and probably non-useful architecture, but an architecture nevertheless.

Even though every system has an architecture, it does not necessarily follow that the architecture is known to anyone. Perhaps all of the people who designed the system are long gone, the documentation

·­·­·- 0 A a

has vanished (or was never produced), the source code has been lost (or was never delivered), and all we have is the executing binary code. This reveals the difference between the architecture of a system and the representation of that architecture. Because an architecture can exist independently of its description or specification, this raises the importance of architecture documentation, which is described in Chapter 18, and architecture reconstruction, discussed in Chapter 20.

Architecture Includes Behavior

The behavior of each element is part of the architecture insofar as that behavior can be used to reason about the system. This behavior embodies how elements interact with each other, which is clearly part of our definition of architecture.

This tells us that box-and-line drawings that are passed off as architectures are in fact not architectures at all. When looking at the names of the boxes (database, graphical user interface, executive, etc.), a reader may well imagine the functionality and behavior of the corresponding elements. This mental image approaches an architecture, but it springs from the imagination of the observer's mind and relies on information that is not present. This does not mean that the exact behavior and performance of every element must be documented in all circumstances some aspects of behavior are fine-grained and below the architect's level of concern. But to the extent that an element's behavior influences another element or influences the acceptability of the system as a whole, this behavior must be considered, and should be documented, as part of the software architecture.

Not All Architectures Are Good Architectures

The definition is indifferent as to whether the architecture for a system is a good one or a bad one. An architecture may permit or preclude a system's achievement of its behavioral, quality attribute, and life­ cycle requirements. Assuming that we do not accept trial and error as the best way to choose an architecture for a system that is, picking an architecture at random, building the system from it, and then hacking away and hoping for the best this raises the importance of architecture design, which is treated in Chapter 17, and architecture evaluation, which we deal with in Chapter 21.

System and Enterprise Architectures

Two disciplines related to software architecture are system architecture and enterprise architecture. Both of these disciplines have broader concerns than software and affect software architecture through the establishment of constraints within which a software system must live. In both cases, the software architect for a system should be on the team that provides input into the decisions made about the system or the enterprise.

System architecture

A system's architecture is a representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and a concern for the human interaction with these components. That is, system architecture is concerned with a total system, including hardware, software, and humans.

A system architecture will determine, for example, the functionality that is assigned

·­·­·- 0 A a

to different processors and the type of network that connects those processors. The software architecture on each of those processors will determine how this functionality is implemented and how the various processors interact through the exchange of messages on the network.

A description of the software architecture, as it is mapped to hardware and networking components, allows reasoning about qualities such as performance and reliability. A description of the system architecture will allow reasoning about additional qualities such as power consumption, weight, and physical footprint.

When a particular system is designed, there is frequently negotiation between the system architect and the software architect as to the distribution of functionality and, consequently, the constraints placed on the software architecture.

Enterprise architecture

Enterprise architecture is a description of the structure and behavior of an organization's processes, information flow, personnel, and organizational subunits, aligned with the organization's core goals and strategic direction. An enterprise architecture need not include information systems clearly organizations had architectures that fit the preceding definition prior to the advent of computers but these days, enterprise architectures for all but the smallest businesses are unthinkable without information system support. Thus, a modem enterprise architecture is concerned with how an enterprise's software systems support the business processes and goals of the enterprise. Typically included in this set of concerns is a process for deciding which systems with which functionality should be supported by an enterprise.

An enterprise architecture will specify the data model that various systems use to interact, for example. It will specify rules for how the enterprise's systems interact with external systems.

Software is only one concern of enterprise architecture. Two other common concerns addressed by enterprise architecture are how the software is used by humans to perform business processes, and the standards that determine the computational environment.

Sometimes the software infrastructure that supports communication among systems and with the external world is considered a portion of the enterprise architecture; other times, this infrastructure is considered one of the systems within an enterprise. (In either case, the architecture of that infrastructure is a software architecture!) These two views will result in different management structures and spheres of influence for the individuals concerned with the infrastructure.

The system and the enterprise provide environments for, and constraints on, the software architecture. The software architecture must live within the system and enterprise, and increasingly it is the focus for achieving the organization's business goals. But all three forms of architecture share important commonalities: They are concerned with major elements taken as abstractions, the relationships among the

A a

·­·­·- 0

System and enterprise architectures share a great deal with software architectures. All can be designed, evaluated, and documented; all answer to requirements; all are intended to satisfy stakeholders; all consist of structures, which in tum consist of elements and relationships; all have a repertoire of patterns and styles at their respective architects' disposal; and the list goes on. So to the extent that these architectures share commonalities with software architecture, they are in the scope of this book. But like all technical disciplines, each has its own specialized vocabulary and techniques, and we won't cover those. Copious other sources do.

elements, and how the elements together meet the behavioral and quality goals of the thing being built.

Are these in scope for this book? Yes! (Well, no.)

·­·­·- 0 A a

1.2. Architectural Structures and Views

The neurologist, the orthopedist, the hematologist, and the dermatologist all have different views of the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on specific subsystems. And the kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior. Although these views are pictured differently and have very different properties, all are inherently related, interconnected: together they describe the architecture of the human body. Figure 1.1 shows several different views of the human body: the skeletal, the vascular, and the X-ray.

,

Figure 1.1. Physiological structures (Getty images: Brand X Pictures [skeleton], Don FarraH [woman], Mads Abildgaard [man])

So it is with software. Modern systems are frequently too complex to grasp all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures. To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment which view we are taking of the architecture.

Structures and Views

We will be using the related terms structure and view when discussing architecture representation.

• A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.

• A structure is the set of elements itself, as they exist in software or hardware.

·­·­·- 0 A a

In short, a view is a representation of a structure. For example, a module structure is the set of the system's modules and their organization. A module view is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders.

So: Architects design structures. They document views of those structures.

Three Kinds of Structures

As we saw in the previous section, architectural structures can be divided into three major categories, depending on the broad nature of the elements they show. These correspond to the three broad kinds of decisions that architectural design involves:

1. Module structures embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured. In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation). Modules represent a static way of considering the system. Modules are assigned areas of functional responsibility; there is less emphasis in these structures on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as these:

• What is the primary functional responsibility assigned to each module?

• What other software elements is a module allowed to use?

• What other software does it actually use and depend on?

• What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

Module structures convey this information directly, but they can also be used by extension to ask questions about the impact on the system when the responsibilities assigned to each module change. In other words, examining a system's module structures that is, looking at its module views is an excellent way to reason about a system's modifiability.

2. Component-and-connector structures embody decisions as to how the system is to be

structured as a set of elements that have runtime behavior (components) and interactions (connectors). In these structures, the elements are runtime components (which are the principal units of computation and could be services, peers, clients, servers, filters, or many other types of runtime elements) and connectors (which are the communication vehicles among components, such as call-return, process synchronization operators, pipes, or others). Component-and-connector views help us answer questions such as these:

• What are the major executing components and how do they interact at runtime?

• What are the major shared data stores?

• Which parts of the system are replicated?

• How does data progress through the system?

• What parts of the system can run in parallel?

• Can the system's structure change as it executes and, if so, how?

·­·­·- 0 A a

By extension, component-and-connector views are crucially important for asking questions about the system's runtime properties such as performance, security, availability, and more.

3. Allocation structures embody decisions as to how the system will relate to nonsoftware

structures in its environment (such as CPUs, file systems, networks, development teams, etc.). These structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation views help us answer questions such as these:

• What processor does each software element execute on?

• In what directories or files is each element stored during development, testing, and system building?

• What is the assignment of each software element to development teams?

Structures Provide Insight

Structures play such an important role in our perspective on software architecture because of the analytical and engineering power they hold. Each structure provides a perspective for reasoning about some of the relevant quality attributes. For example:

• The module "uses" structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted.

• The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks.

• The deployment structure is strongly tied to the achievement of performance, availability, and security goals.

And so forth. Each structure provides the architect with a different insight into the design (that is, each structure can be analyzed for its ability to deliver a quality attribute). But perhaps more important, each structure presents the architect with an engineering leverage point: By designing the structures appropriately, the desired quality attributes emerge.

Scenarios, described in Chapter 4, are useful for exercising a given structure as well as its connections to other structures. For example, a software engineer wanting to make a change to the concurrency structure of a system would need to consult the concurrency and deployment views, because the affected mechanisms typically involve processes and threads, and physical distribution might involve different control mechanisms than would be used if the processes were co-located on a single machine. If control mechanisms need to be changed, the module decomposition would need to be consulted to determine the extent of the changes.

Some Useful Module Structures

Useful module structures include the following:

• Decomposition structure. The units are modules that are related to each other by the is-a­ submodule-of relation, showing how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood. Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will

·­·­·- 0 A a

have to do and assigns each item to a module for subsequent (more detailed) design and eventual implementation. Modules often have products (such as interface specifications, code, test plans, etc.) associated with them. The decomposition structure determines, to a large degree, the system's modifiability, by assuring that likely changes are localized. That is, changes fall within the purview of at most a few (preferably small) modules. This structure is often used as the basis for the development project's organization, including the structure of the documentation, and the project's integration and test plans. The units in this structure tend to have names that are organization-specific such as "segment" or "subsystem."

• Uses structure. In this important but overlooked structure, the units here are also modules, perhaps classes. The units are related by the uses relation, a specialized form of dependency. A unit of software uses another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second. The uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted. The ability to easily create a subset of a syste1n allows for incremental development.

• Layer structure. The modules in this structure are called layers. A layer is an abstract "virtual machine" that provides a cohesive set of services through a managed interface. Layers are allowed to use other layers in a strictly managed fashion; in strictly layered systems, a layer is only allowed to use the layer immediately below. This structure is used to imbue a system with portability, the ability to change the underlying computing platform.

• Class (or generalization) structure. The module units in this structure are called classes. The relation is inherits from or is an instance of This view supports reasoning about collections of similar behavior or capability (e.g., the classes that other classes inherit from) and parameterized differences. The class structure allows one to reason about reuse and the incremental addition of functionality. If any documentation exists for a project that has followed an object-oriented analysis and design process, it is typically this structure.

• Data model. The data model describes the static information structure in tenns of data entities and their relationships. For example, in a banking system, entities will typically include Account, Customer, and Loan. Account has several attributes, such as account number, type (savings or checking), status, and current balance. A relationship may dictate that one custo·mer can have one or 1nore accounts, and one account is associated to one or two customers.

Some Useful C&C Structures

Component-and-connector structures show a runtime view of the system. In these structures the modules described above have all been compiled into executable forms. All component-and-connector structures are thus orthogonal to the module-based structures and deal with the dynamic aspects of a running syste1n. The relation in all component-and-connector structures is attachment, showing how the components and the connectors are hooked together. (The connectors themselves can be familiar constructs such as "invokes.") Useful C&C structures include the following:

• Service structure. The units here are services that interoperate with each other by service coordination mechanisms such as SOAP (see Chapter 6). The service structure is an important

·­·­·- 0 A a

structure to help engineer a system composed of components that may have been developed anonymously and independently of each other.

• Concurrency structure. This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are their communication mechanisms. The components are arranged into logical threads; a logical thread is a sequence of computations that could be allocated to a separate physical thread later in the design process. The concurrency structure is used early in the design process to identify the requirements to manage the issues associated with concurrent execution.

Some Useful Allocation Structures

Allocation structures define how the elements from C&C or module structures map onto things that are not software: typically hardware, teams, and file systems. Useful allocation structures include these:

• Deployment structure. The deployment structure shows how software is assigned to hardware processing and communication elements. The elements are software elements (usually a process from a C&C view), hardware entities (processors), and communication pathways. Relations are allocated-to, showing on which physical units the software elements reside, and migrates-to if the allocation is dynamic. This structure can be used to reason about performance, data integrity, security, and availability. It is of particular interest in distributed and parallel systems.

• Implementation structure. This structure shows how software elements (usually modules) are mapped to the file structure( s) in the system's development, integration, or configuration control environments. This is critical for the management of development activities and build processes. (In practice, a screenshot of your development environment tool, which manages the implementation environment, often makes a very useful and sufficient diagram of your implementation view.)

• Work assignment structure. This structure assigns responsibility for implementing and integrating the 1nodules to the teams who will carry it out. Having a work assignment structure be part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications. The architect will know the expertise required on each team. Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning those to a single team, rather than having them implemented by everyone who needs them. This structure will also determine the major communication pathways among the teams: regular teleconferences, wikis, email lists, and so forth.

Table 1.1 summarizes these structures. The table lists the meaning of the elements and relations in each structure and tells what each might be used for.

Table 1.1. Useful Architectural Structures

A a

Each of these structures provides a different perspective and design handle on a system, and each is valid and useful in its own right. Although the structures give different system perspectives, they are not independent. Elements of one structure will be related to elements of other structures, and we need to reason about these relations. For example, a module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures, reflecting its runtime alter ego. In general, mappings between structures are many to many.

Figure 1.2 shows a very simple example of how two structures might relate to each other. The figure on the left shows a module decomposition view of a tiny client-server system. In this system, two modules must be implemented: The client software and the server software. The figure on the right shows a component-and-connector view of the same system. At runtime there are ten clients running and accessing the server. Thus, this little system has two modules and eleven components (and ten connectors).

System

Client

Server

Key: J Module I

Decomposttion View

0 Quality Attributes

Affected

Modifiability

"Subsetability; extensibility

Portability

Class Class. object Is an instance of. shares access In object-oriented design systems. factoring out Modifiability.

extensibility

C&C Structures

Allocation Structures

methods of

commonality; planning extensions of functionality

Data model

Data entity {one, many}-to-{one, many].

Engineering global dala structures for consistency

Modifiability. generalizes, specializes

and performance

performance

Service

Service, ESB, registry, Runs concurrently with, may

Scheduling analysis, performance analysis

lnleroperability, others run concurrently with. excludes.

modifiability precedes, etc.

Concurrency

Processes, threads Can run In parallel

Identifying locations where resource contention

Per lonna nee, exists. or where threads may fork, join, be created.

availabifijy or be killed

Deployment

Components, hardware Allocated to, migrates to

Performance, availability. security analysis

Perfonnance. elements

security, availabifijy

Implementation Modules. me structure Slored in

Configuration control, integration, test activities Development

efficiency

Work assignment Modules, organizational Assigned to

Projecl managemenl, best use of expertise and Development

units

available resources. management of commonality efficiency

Relating Structures to Each Other

Key: Q· Component

C5

- Request-Reply

Client-Server Vi,ew

Figure 1.2. Two views of a client-server system

Whereas the correspondence between the elements in the decomposition structure and the client­ server structure is obvious, these two views are used for very different things. For example, the view on

Software Element Structure Types Relations

Module Decomposilion Module Is a submodufe of Structures

Uses Module Uses (i.e .• requires the corree1

presence of)

Layers Layer Requires the correct presence

of, uses the services of, provides abstraction to

·­·­·-

Useful For

Resource allocalion and pro1ect structuring and planning; information hiding, encapsulation; configuration control

Engineering subsets. engineering extensions

Incremental development, implementing systems on top of "virtual machines"

·­·­·- 0 A a

the right could be used for performance analysis, bottleneck prediction, and network traffic managetnent, which would be extremely difficult or impossible to do with the view on the left.

(In Chapter 13 we'll learn about the map-reduce pattern, in which copies of simple, identical functionality are distributed across hundreds or thousands of processing nodes one module for the whole system, but one component per node.)

Individual projects sometimes consider one structure dominant and cast other structures, when possible, in tenns of the dominant structure. Often the dominant structure is the module decomposition structure. This is for a good reason: it tends to spawn the project structure, because it mirrors the team structure of development. In other projects, the dominant structure might be a C&C structure that shows how the system's functionality and/or critical quality attributes are achieved.

Fewer Is Better

Not all systems warrant consideration of many architectural structures. The larger the systetn, the more dramatic the difference between these structures tends to be; but for small systems we can often get by with fewer. Instead of working with each of several component-and-connector structures, usually a single one will do. If there is only one process, then the process structure collapses to a single node and need not be explicitly represented in the design. If there is to be no distribution (that is, if the system is implemented on a single processor), then the deployment structure is trivial and need not be considered further. In general, design and document a structure only if doing so brings a positive return on the investment, usually in tenns of decreased development or maintenance costs.

Which Structures to Choose?

We have briefly described a number of useful architectural structures, and there are many more. Which ones shall an architect choose to work on? Which ones shall the architect choose to document? Surely not all of them. Chapter 18 will treat this topic in more depth, but for now a good answer is that you should think about how the various structures available to you provide insight and leverage into the system's most important quality attributes, and then choose the ones that will play the best role in delivering those attributes.

Ask Cal

More than a decade ago I went to a customer site to do an architecture evaluation one of the first instances of the Architecture Tradeoff Analysis Method (AT AM) that I had ever performed (you can read about the AT AM, and other architecture evaluation topics, in Chapter 21). In those early days, we were still figuring out how to make architecture evaluations repeatable and predictable, and how to guarantee useful outcomes from them. One of the ways that we ensured useful outco1nes was to enforce certain preconditions on the evaluation. A precondition that we figured out rather quickly was this: if the architecture has not been documented, we will not proceed with the evaluation. The reason for this precondition was simple: we could not evaluate the architecture by reading the code we didn't have the time for that and we couldn't just ask the architect to sketch the architecture in real time, since that would produce

A ·­·­·- 0

a

vague and very likely erroneous representations.

Okay, it's not completely true to say that they had no architecture documentation. They did produce a single-page diagram, with a few boxes and lines. Some of those boxes were, however, clouds. Yes, they actually used a cloud as one of their icons. When I pressed them on the meaning of this icon Was it a process? A class? A thread? they waffled. This was not, in fact, architecture documentation. It was, at best, "marketecture."

But in those early days we had no preconditions and so we didn't stop the evaluation there. We just blithely waded in to whatever swamp we found, and we enforced nothing. As I began this evaluation, I interviewed some of the key project stakeholders: the project manager and several of the architects (this was a large project with one lead architect and several subordinates). As it happens, the lead architect was away, and so I spent my time with the subordinate architects. Every time I asked the subordinates a tough question "How do you ensure that you will meet your latency goal along this critical execution path?" or "What are your rules for layering?" they would answer: "Ask Cal. Cal knows that." Cal was the lead architect. Immediately I noted a risk for this system: What if Cal gets hit by a bus? What then?

In the end, because of my pestering, the architecture team did in fact produce respectable architecture documentation. About halfway through the evaluation, the project manager came up to me and shook my hand and thanked me for the great job I had done. I was dumbstruck. In my mind I hadn't done anything, at that point; the evaluation was only partially complete and I hadn't produced a single report or finding. I said that to the manager and he said: "You got those guys to document the architecture. I've never been able to get them to do that. So . . . thanks!"

If Cal had been hit by a bus or just left the company, they would have had a serious problem on their hands: all of that architectural knowledge located in one guy's head and he is no longer with the organization. In can happen. It does happen.

The moral of this story? An architecture that is not documented, and not communicated, may still be a good architecture, but the risks surrounding it are enormous.

-RK

·­·­·- 0 A a

1.3. Architectural Patterns

In some cases, architectural elements are composed in ways that solve particular problems. The compositions have been found useful over time, and over many different domains, and so they have been documented and disseminated. These compositions of architectural elements, called architectural patterns, provide packaged strategies for solving some of the problems facing a system.

An architectural pattern delineates the element types and their forms of interaction used in solving the problem. Patterns can be characterized according to the type of architectural elements they use. For example, a common module type pattern is this:

• Layered pattern. When the uses relation among software elements is strictly unidirectional, a system of layers emerges. A layer is a coherent set of related functionality. In a strictly layered structure, a layer can only use the services of the layer immediately below it. Many variations of this patten1, lessening the structural restriction, occur in practice. Layers are often designed as abstractions (virtual tnachines) that hide implementation specifics below from the layers above, engendering portability.

Common component-and-connector type patterns are these:

• Shared-data (or repository) pattern. This pattern comprises components and connectors that create, store, and access persistent data. The repository usually takes the form of a (commercial) database. The connectors are protocols for managing the data, such as SQL.

• Client-server pattern. The components are the clients and the servers, and the connectors are protocols and messages they share among each other to carry out the system's work.

Common allocation patterns include the following:

• Multi-tier pattern, which describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium. This pattern specializes the generic deployment (software-to-hardware allocation) structure.

• Competence center and platform, which are patterns that specialize a software system's work assignment structure. In competence center, work is allocated to sites depending on the technical or domain expertise located at a site. For example, user-interface design is done at a site where usability engineering experts are located. In platform, one site is tasked with developing reusable core assets of a software product line (see Chapter 25), and other sites develop applications that use the core assets.

Architectural patterns will be investigated much further in Chapter 13.

·­·­·- 0 A a

1.4. What Makes a "Good" Architecture?

There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose. A three-tier layered service-oriented architecture may be just the ticket for a large enterprise's web-based B2B system but completely wrong for an avionics application. An architecture carefully crafted to achieve high modifiability does not make sense for a throwaway prototype (and vice versa!). One of the messages of this book is that architectures can in fact be evaluated one of the great benefits of paying attention to them but only in the context of specific stated goals.

Nevertheless, there are rules of thumb that should be followed when designing most architectures. Failure to apply any of these does not automatically mean that the architecture will be fatally flawed, but it should at least serve as a warning sign that should be investigated.

We divide our observations into two clusters: process recommendations and product (or structural) recommendations. Our process recommendations are the following:

1. The architecture should be the product of a single architect or a small group of architects with

an identified technical leader. This approach gives the architecture its conceptual integrity and technical consistency. This recommendation holds for Agile and open source projects as well as "traditional" ones. There should be a strong connection between the architect(s) and the development team, to avoid ivory tower designs that are impractical.

2. The architect (or architecture team) should, on an ongoing basis, base the architecture on a

prioritized list of well-specified quality attribute requirements. These will inform the tradeoffs that always occur. Functionality matters less.

3. The architecture should be documented using views. The views should address the concerns

of the most important stakeholders in support of the project timeline. This might mean minimal documentation at first, elaborated later. Concerns usually are related to construction, analysis, and maintenance of the system, as well as education of new stakeholders about the system.

4. The architecture should be evaluated for its ability to deliver the system's important quality attributes. This should occur early in the life cycle, when it returns the most benefit, and repeated as appropriate, to ensure that changes to the architecture (or the environment for which it is intended) have not rendered the design obsolete.

5. The architecture should lend itself to incremental implementation, to avoid having to

integrate everything at once (which almost never works) as well as to discover problems early. One way to do this is to create a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality. This skeletal system can be used to "grow" the system incrementally, refactoring as necessary.

Our structural rules of thumb are as follows:

1. The architecture should feature well-defmed modules whose functional responsibilities are

assigned on the principles of information hiding and separation of concerns. The information­ hiding modules should encapsulate things likely to change, thus insulating the software from the effects of those changes. Each module should have a well-defined interface that

·­·­·- 0 A a

encapsulates or "hides" the changeable aspects from other software that uses its facilities. These interfaces should allow their respective development teams to work largely independently of each other.

2. Unless your requirements are unprecedented possible, but unlikely your quality attributes should be achieved using well-known architectural patterns and tactics (described in Chapter 13) specific to each attribute.

3. The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and

• • Inexpensive.

4. Modules that produce data should be separate from modules that consume data. This tends to increase modifiability because changes are frequently confined to either the production or the consumption side of data. If new data is added, both sides will have to change, but the separation allows for a staged (incremental) upgrade.

5. Don't expect a one-to-one correspondence between modules and components. For example, in systems with concurrency, there may be multiple instances of a component running in parallel, where each component is built from the same module. For systems with multiple threads of concurrency, each thread Inay use services from several c01nponents, each of which was built fr01n a different module.

6. Every process should be written so that its assignment to a specific processor can be easily

changed, perhaps even at runtime.

7. The architecture should feature a small number of ways for components to interact. That is,

the system should do the same things in the same way throughout. This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.

8. The architecture should contain a specific (and small) set of resource contention areas, the

resolution of which is clearly specified and maintained. For example, if network utilization is an area of concern, the architect should produce (and enforce) for each development team guidelines that will result in a minimum of network traffic. If performance is a concern, the architect should produce (and enforce) time budgets for the major threads.

·­·­·- 0 A a

1.5. Summary

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

A structure is a set of elements and the relations among them.

A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. A view is a representation of one or more structures.

There are three categories of structures:

• Module structures show how a system is to be structured as a set of code or data units that have to be constructed or procured.

• Component-and-connector structures show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).

• Allocation structures show how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.).

Structures represent the primary engineering leverage points of an architecture. Each structure brings with it the power to manipulate one or more quality attributes. They represent a powerful approach for creating the architecture (and later, for analyzing it and explaining it to its stakeholders). And as we will see in Chapter 18, the structures that the architect has chosen as engineering leverage points are also the primary candidates to choose as the basis for architecture documentation.

Every system has a software architecture, but this architecture may be documented and disseminated, or it may not be.

There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose.

·­·­·- 0 A a

The early work of David Parnas laid much of the conceptual foundation for what became the study of software architecture. A quintessential Parnas reader would include his foundational article on information hiding [Parnas 72] as well as his works on program families [Parnas 76], the structures inherent in software systems [Parnas 74], and introduction of the uses structure to build subsets and supersets of systems [Parnas 79]. All of these papers can be found in the more easily accessible collection of his important papers [Hoffman 00].

An early paper by Perry and Wolf [Perry 92] drew an analogy between software architecture views and structures and the structures one finds in a house (plumbing, electrical, and so forth).

Software architectural patterns have been extensively catalogued in the series Pattern-Oriented Software Architecture [Buschmann 96] and others. Chapter 13 of this book also deals with architectural patterns.

Early papers on architectural views as used in industrial development projects are [Soni 95] and [Kruchten 95]. The former grew into a book [Hofmeister 00] that presents a comprehensive picture of using views in development and analysis. The latter grew into the Rational Unified Process, about which there is no shortage of references, both paper and online. A good one is [Kruchten 03].

Cristina Gacek and her colleagues discuss the process issues surrounding software architecture in [Gacek 95].

Garlan and Shaw's seminal work on software architecture [Garlan 93] provides many excellent examples of architectural styles (a concept similar to patterns).

In [Clements 1 Oa] you can find an extended discussion on the difference between an architectural pattern and an architectural style. (It argues that a pattern is a context-problem-solution triple; a style is simply a condensation that focuses most heavily on the solution part.)

See [Taylor 09] for a definition of software architecture based on decisions rather than on structure.

1.6. For Further Reading

·­·­·- 0 A a

1.7. Discussion Questions

1. Software architecture is often compared to the architecture of buildings as a conceptual analogy.

What are the strong points of that analogy? What is the correspondence in buildings to software architecture structures and views? To patterns? What are the weaknesses of the analogy? When does it break down?

2. Do the architectures you've been exposed to document different structures and relations like

those described in this chapter? If so, which ones? If not, why not?

3. Is there a different definition of software architecture that you are familiar with? If so, compare and contrast it with the definition given in this chapter. Many definitions include considerations like "rationale" (stating the reasons why the architecture is what it is) or how the architecture will evolve over time. Do you agree or disagree that these considerations should be part of the definition of software architecture?

4. Discuss how an architecture serves as a basis for analysis. What about decision-making? What

kinds of decision-making does an architecture empower?

5. What is architecture's role in project risk reduction?

6. Find a commonly accepted definition of system architecture and discuss what it has in common

with software architecture. Do the same for enterprise architecture.

7. Find a published example of an architecture. What structure or structures are shown? Given its

purpose, what structure or structures should have been shown? What analysis does the architecture support? Critique it: What questions do you have that the representation does not answer?

8. Sailing ships have architectures, which means they have "structures" that lend themselves to reasoning about the ship's performance and other quality attributes. Look up the technical definitions for barque, brig, cutter,ji�igate, ketch, schooner, and sloop. Propose a useful set of "structures" for distinguishing and reasoning about ship architectures.

A a

0

·­·­·-

·­·­·-

While Chapter 3 will cover the business importance of architecture to an enterprise, this chapter focuses on why architecture matters from a technical perspective. We will examine a baker's dozen of the most important reasons.

1. An architecture will inhibit or enable a system's driving quality attributes.

2. The decisions made in an architecture allow you to reason about and manage change as the

system evolves.

3. The analysis of an architecture enables early prediction of a system's qualities.

4. A documented architecture enhances communication among stakeholders.

5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change

design decisions.

6. An architecture defines a set of constraints on subsequent implementation.

7. The architecture dictates the structure of an organization, or vice versa.

8. An architecture can provide the basis for evolutionary prototyping.

9. An architecture is the key artifact that allows the architect and project manager to reason

about cost and schedule.

10. An architecture can be created as a transferable, reusable model that forms the heart of a

product line.

11. Architecture-based development focuses attention on the assembly of components, rather

than simply on their creation.

12. By restricting design alternatives, architecture channels the creativity of developers, reducing

design and system complexity.

13. An architecture can be the foundation for training a new team member.

Even if you already believe us that architecture is important and don't need the point hammered thirteen more titnes, think of these thirteen points (which form the outline for this chapter) as thirteen useful ways to use architecture in a project.

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

If architecture is the answer, what was the question?

2. Why Is Software Architecture Important?

0 A a

·­·­·-

Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.

This is such an important message that we've devoted all of Part 2 of this book to expounding that message in detail. Until then, keep these examples in mind as a starting point:

• If your system requires high performance, then you need to pay attention to managing the time­ based behavior of elements, their use of shared resources, and the frequency and volume of inter -element comtnunication.

• If modifiability is important, then you need to pay careful attention to assigning responsibilities to elements so that the majority of changes to the system will affect a small number of those elements. (Ideally each change will affect just a single element.)

• If your system must be highly secure, then you need to manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism) into the architecture.

• If you believe that scalability will be important to the success of your system, then you need to carefully localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hard-coding in resource assumptions or limits.

• If your projects need the ability to deliver incremental subsets of the system, then you must carefully tnanage intercomponent usage.

• If you want the elements from your system to be reusable in other systems, then you need to restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment to be useful.

The strategies for these and other quality attributes are supremely architectural. But an architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions can always undermine an adequate architectural design. As we like to say (mostly in jest): The architecture giveth and the implementation taketh away. Decisions at all stages of the life cycle from architectural design to coding and implementation affect system quality. Therefore, quality is not completely a function of an architectural design.

A good architecture is necessary, but not sufficient, to ensure quality. Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct.

For example, modifiability is determined by how functionality is divided and coupled (architectural) and by coding techniques within a module (nonarchitectural). Thus, a system is typically modifiable if changes involve the fewest possible number of distinct elements. In spite of having the ideal architecture, however, it is always possible to make a system difficult to modify by writing obscure, tangled code.

2.1. Inhibiting or Enabling a System's Quality Attributes

·­·­·- 0 A a

2.2. Reasoning About and Managing Change

This point is a corollary to the previous point.

Modifiability the ease with which changes can be made to a system is a quality attribute (and hence covered by the arguments in the previous section), but it is such an important quality that we have awarded it its own spot in the List of Thirteen. The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's total cost occurs after initial deployment. A corollary of this statistic is that most systems that people work on are in this phase. Many programmers and software designers never get to work on new development; they work under the constraints of the existing architecture and the existing body of code. Virtually all software systems change over their lifetime, to accommodate new features, to adapt to new environments, to fix bugs, and so forth. But these changes are often fraught with difficulty.

Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.

• A local change can be accomplished by modifying a single element. For example, adding a new business rule to a pricing logic module.

• A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact. For example, adding a new business rule to a pricing logic module, then adding new fields to the database that this new business rule requires, and then revealing the results of the rule in the user interface.

• An architectural change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system. For example, changing a system from client-server to peer-to-peer.

Obviously, local changes are the most desirable, and so an effective architecture is one in which the most common changes are local, and hence easy to make.

Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, performance, and behaviors of system software elements. These activities are in the job description for an architect. Reasoning about the architecture and analyzing the architecture can provide the insight necessary to make decisions about anticipated changes.

·­·­·- 0 A a

2.3. Predicting System Qualities

This point follows from the previous two. Architecture not only imbues systems with qualities, but it does so in a predictable way.

Were it not possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed, then choosing an architecture would be a hopeless task randomly making architecture selections would perform as well as any other method. Fortunately, it is possible to make quality predictions about a system based solely on an evaluation of its architecture. If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, then we can make those decisions and rightly expect to be rewarded with the associated quality attributes. After the fact, when we examine an architecture, we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.

This is no different from any mature engineering discipline, where design analysis is a standard part of the development process. The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Even if you don't do the quantitative analytic modeling sometimes necessary to ensure that an architecture will deliver its prescribed benefits, this principle of evaluating decisions based on their quality attribute implications is invaluable for at least spotting potential trouble spots early.

The architecture modeling and analysis techniques described in Chapter 14, as well as the architecture evaluation teclmiques covered in Chapter 21, allow early insight into the software product qualities made possible by software architectures.

·­·­·- 0 A a

2.4. Enhancing Communication among Stakeholders

Software architecture represents a common abstraction of a system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, negotiating, forming consensus, and communicating with each other. The architecture or at least parts of it is sufficiently abstract that most nontechnical people can understand it adequately, particularly with some coaching from the architect, and yet that abstraction can be refined into sufficiently rich technical specifications to guide implementation, integration, test, and deployment.

Each stakeholder of a software system customer, user, project manager, coder, tester, and so on­ is concerned with different characteristics of the system that are affected by its architecture. For example:

• The user is concerned that the system is fast, reliable, and available when needed.

• The customer is concerned that the architecture can be implemented on schedule and according to budget.

• The manager is worried (in addition to concerns about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.

• The architect is worried about strategies to achieve all of those goals.

Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness. Architectural analysis, as we will see in Chapter 21, both depends on this level of communication and enhances it.

Section 3.5 covers stakeholders and their concerns in greater depth.

"What Happens When I Push This Button?" Architecture as a Vehicle for

Stakeholder Communication

The project review droned on and on. The government-sponsored development was behind schedule and over budget and was large enough that these lapses were attracting congressional attention. And now the government was making up for past neglect by holding a marathon come-one-come-all review session. The contractor had recently undergone a buyout, which hadn't helped matters. It was the afternoon of the second day, and the agenda called for the software architecture to be presented. The young architect an apprentice to the chief architect for the system was bravely explaining how the software architecture for the massive system would enable it to meet its very demanding real-time, distributed, high-reliability requirements. He had a solid presentation and a solid architecture to present. It was sound and sensible. But the audience about 30 government representatives who had varying roles in the management and oversight of this sticky project was tired. Some of them were even thinking that perhaps they should have gone into real estate instead of enduring another one of these marathon let's-finally-get-it-right-this-time reviews.

·­·­·- 0 A a

The viewgraph showed, in semiformal box-and-line notation, what the major software elements were in a runtime view of the system. The names were all acronyms, suggesting no setnantic meaning without explanation, which the young architect gave. The lines showed data flow, message passing, and process synchronization. The elements were internally redundant, the architect was explaining. "In the event of a failure," he began, using a laser pointer to denote one of the lines, "a restart mechanism triggers along this path when "

"What happens when the mode select button is pushed?" interrupted one of the audience members. He was a government attendee representing the user community for this system.

"Beg your pardon?" asked the architect.

"The mode select button," he said. "What happens when you push it?"

"Urn, that triggers an event in the device driver, up here," began the architect, laser­ pointing. "It then reads the register and interprets the event code. If it's mode select, well, then, it signals the blackboard, which in turns signals the objects that have subscribed to that event. . . . "

"No, I mean what does the system do," interrupted the questioner. "Does it reset the displays? And what happens if this occurs during a system reconfiguration?"

The architect looked a little surprised and flicked off the laser pointer. This was not an architectural question, but since he was an architect and therefore fluent in the requirements, he knew the answer. "If the command line is in setup mode, the displays will reset," he said. "Otherwise an error message will be put on the control console, but the signal will be ignored." He put the laser pointer back on. "Now, the restart mechanism that I was talking about "

"Well, I was just wondering," said the users' delegate. "Because I see from your chart that the display console is sending signal traffic to the target location module."

"What should happen?" asked another member of the audience, addressing the first questioner. "Do you really want the user to get mode data during its reconfiguring?" And for the next 45 minutes, the architect watched as the audience consumed his time slot by debating what the correct behavior of the system was supposed to be in various esoteric states.

The debate was not architectural, but the architecture (and the graphical rendition of it) had sparked debate. It is natural to think of architecture as the basis for communication among some of the stakeholders besides the architects and developers: Managers, for example, use the architecture to create teams and allocate resources among them. But users? The architecture is invisible to users, after all; why should they latch on to it as a tool for understanding the system?

The fact is that they do. In this case, the questioner had sat through two days of viewgraphs all about function, operation, user interface, and testing. But it was the first

·­·­·- 0 A a

slide on architecture that even though he was tired and wanted to go home made him realize he didn't understand something. Attendance at many architecture reviews has convinced me that seeing the system in a new way prods the mind and brings new questions to the surface. For users, architecture often serves as that new way, and the questions that a user poses will be behavioral in nature. In a memorable architecture evaluation exercise a few years ago, the user representatives were much more interested in what the system was going to do than in how it was going to do it, and naturally so. Up until that point, their only contact with the vendor had been through its marketers. The architect was the first legitimate expert on the system to whom they had access, and they didn't hesitate to seize the moment.

Of course, careful and thorough requirements specifications would ameliorate this situation, but for a variety of reasons they are not always created or available. In their absence, a specification of the architecture often serves to trigger questions and improve clarity. It is probably more prudent to recognize this reality than to resist it.

Sometimes such an exercise will reveal unreasonable requirements, whose utility can then be revisited. A review of this type that emphasizes synergy between requirements and architecture would have let the young architect in our story off the hook by giving him a place in the overall review session to address that kind of information. And the user representative wouldn't have felt like a fish out of water, asking his question at a clearly inappropriate moment.

-PCC

·­·­·- 0 A a

2.5. Carrying Early Design Decisions

Software architecture is a manifestation of the earliest design decisions about a system, and these early bindings carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life. It is also the earliest point at which these important design decisions affecting the system can be scrutinized.

Any design, in any discipline, can be viewed as a set of decisions. When painting a picture, an artist decides on the material for the canvas, on the media for recording oil paint, watercolor, crayon even before the picture is begun. Once the picture is begun, other decisions are immediately made: Where is the first line? What is its thickness? What is its shape? All of these early design decisions have a strong influence on the final appearance of the picture. Each decision constrains the many decisions that follow. Each decision, in isolation, might appear innocent enough, but the early ones in particular have disproportionate weight simply because they influence and constrain so much of what follows.

So it is with architecture design. An architecture design can also be viewed as a set of decisions. The early design decisions constrain the decisions that follow, and changing these decisions has enormous ramifications. Changing these early decisions will cause a ripple effect, in terms of the additional decisions that must now be changed. Yes, sometimes the architecture must be refactored or redesigned, but this is not a task we undertake lightly (because the "ripple" might turn into a tsunami).

What are these early design decisions embodied by software architecture? Consider:

• Will the system run on one processor or be distributed across multiple processors?

• Will the software be layered? If so, how many layers will there be? What will each one do?

• Will components communicate synchronously or asynchronously? Will they interact by transferring control or data or both?

• Will the system depend on specific features of the operating system or hardware?

• Will the information that flows through the syste1n be encrypted or not?

• What operating system will we use?

• What communication protocol will we choose?

Imagine the nightmare of having to change any of these or a myriad other related decisions. Decisions like these begin to flesh out some of the structures of the architecture and their interactions. In Chapter 4, we describe seven categories of these early design decisions. In Chapters 5-1 1 we show the implications of these design decision categories on achieving quality attributes.

·­·­·- 0 A a

2.6. Defining Constraints on an Implementation

An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture. This means that the implementation must be implemented as the set of prescribed elements, these elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the other elements as dictated by the architecture. Each of these prescriptions is a constraint on the implementer.

Element builders must be fluent in the specifications of their individual elements, but they may not be aware of the architectural tradeoffs the architecture (or architect) simply constrains them in such a way as to meet the tradeoffs. A classic example of this phenomenon is when an architect assigns performance budget to the pieces of software involved in some larger piece of functionality. If each software unit stays within its budget, the overall transaction will meet its performance requirement. Implementers of each of the constituent pieces may not know the overall budget, only their own.

Conversely, the architects need not be experts in all aspects of algorithm design or the intricacies of the programtning language although they should certainly know enough not to design something that is difficult to build but they are the ones responsible for establishing, analyzing, and enforcing the architectural tradeoffs.

·­·­·- 0 A a

2. 7. Influencing the Organizational Structure

Not only does architecture prescribe the structure of the system being developed, but that structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization). The normal method for dividing up the labor in a large project is to assign different groups different portions of the system to construct. This is called the work-breakdown structure of a system. Because the architecture includes the broadest decomposition of the system, it is typically used as the basis for the work-breakdown structure. The work-breakdown structure in turn dictates units of planning, scheduling, and budget; interteam communication channels; configuration control and file­ system organization; integration and test plans and procedures; and even project minutiae such as how the project intranet is organized and who sits with whom at the company picnic. Teams communicate with each other in terms of the interface specifications for the major elements. The maintenance activity, when launched, will also reflect the software structure, with teams fonned to maintain specific structural elements from the architecture: the database, the business rules, the user interface, the device drivers, and so forth.

A side effect of establishing the work-breakdown structure is to freeze some aspects of the software architecture. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive or even litigious.

Thus, once the architecture has been agreed on, it becomes very costly for managerial and business reasons to significantly modify it. This is one argument (among many) for carrying out extensive analysis before settling on the software architecture for a large system because so much depends on it.

·­·­·- 0 A a

2.8. Enabling Evolutionary Prototyping

Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. A skeletal system is one in which at least some of the infrastructure how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth is built before much of the system's functionality has been created. (The two can go hand in hand: build a little infrastructure to support a little end-to-end functionality; repeat until done.)

For example, systems built as plug-in architectures are skeletal systems: the plug-ins provide the actual functionality. This approach aids the development process because the system is executable early in the product's life cycle. The fidelity of the system increases as stubs are instantiated, or prototype parts are replaced with complete versions of these parts of the software. In some cases the prototype parts can be low-fidelity versions of the final functionality, or they can be surrogates that consume and produce data at the appropriate rates but do little else. Among other things, this approach allows potential performance problems to be identified early in the product's life cycle.

These benefits reduce the potential risk in the project. Furthermore, if the architecture is part of a family of related systems, the cost of creating a framework for prototyping can be distributed over the development of many systems.

·­·­·- 0 A a

2.9. Improving Cost and Schedule Estimates

Cost and schedule estimates are important tools for the project manager both to acquire the necessary resources and to monitor progress on the project, to know if and when a project is in trouble. One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle. Although top-down estimates are useful for setting goals and apportioning budgets, cost estimations that are based on a bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.

As we have said, the organizational and work-breakdown structure of a project is almost always based on its architecture. Each team or individual responsible for a work item will be able to make more-accurate estimates for their piece than a project manager and will feel more ownership in making the estimates come true. But the best cost and schedule estimates will typically emerge from a consensus between the top-down estimates (created by the architect and project manager) and the bottom-up estimates (created by the developers). The discussion and negotiation that results from this process creates a far more accurate estimate than either approach by itself.

It helps if the requirements for a system have been reviewed and validated. The more up-front knowledge you have about the scope, the more accurate the cost and schedule estimates will be.

Chapter 22 delves into the use of architecture in project management.

·­·­·- 0 A a

2.1 0. Supplying a Transferable, Reusable Model

The earlier in the life cycle that reuse is applied, the greater the benefit that can be achieved. While code reuse provides a benefit, reuse of architectures provides tremendous leverage for systems with similar requirements. Not only can code be reused, but so can the requirements that led to the architecture in the first place, as well as the experience and infrastructure gained in building the reused architecture. When architectural decisions can be reused across multiple systems, all of the early-decision consequences we just described are also transferred.

A software product line or family is a set of software systems that are all built using the same set of reusable assets. Chief among these assets is the architecture that was designed to handle the needs of the entire family. Product-line architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line. The architecture defines what is fixed for all members of the product line and what is variable. Software product lines represent a powerful approach to multi-system development that is showing order-of-magnitude payoffs in time to market, cost, productivity, and product quality. The power of architecture lies at the heart of the paradigm. Similar to other capital investments, the architecture for a product line becomes a developing organization's core asset. Software product lines are explained in Chapter 25.

A ·­·­·- 0

a

2.11. Allowing Incorporation of Independently Developed Components

Whereas earlier software paradigms have focused on programming as the prime activity, with progress measured in lines of code, architecture-based development often focuses on composing or assembling elements that are likely to have been developed separately, even independently, from each other. This composition is possible because the architecture defines the elements that can be incorporated into the system. The architecture constrains possible replacements (or additions) according to how they interact with their environment, how they receive and relinquish control, what data they consume and produce, how they access data, and what protocols they use for communication and resource sharing.

In 1793, Eli Whitney's mass production of muskets, based on the principle of interchangeable parts, signaled the dawn of the industrial age. In the days before physical measurements were reliable, manufacturing interchangeable parts was a daunting notion. Today in software, until abstractions can be reliably delimited, the notion of structural interchangeability is just as daunting and just as significant.

Commercial off-the-shelf components, open source software, publicly available apps, and networked services are all modem-day software instantiations of Whitney's basic idea. Whitney's musket parts had "interfaces" (having to do with fit and durability) and so do today's interchangeable software components.

For software, the payoff can be

• Decreased time to market (it should be easier to use someone else's ready solution than build your own)

• Increased reliability (widely used software should have its bugs ironed out already)

• Lower cost (the software supplier can amortize development cost across their customer base)

• Flexibility (if the component you want to buy is not terribly special-purpose, it's likely to be available from several sources, thus increasing your buying leverage)

·­·­·- 0 A a

2.12. Restricting the Vocabulary of Design Alternatives

As useful architectural patterns are collected, it becomes clear that although software elements can be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small number of choices of elements and their interactions. By doing so we minimize the design complexity of the system we are building.

A software engineer is not an artiste, whose creativity and freedom are paramount. Engineering is about discipline, and discipline comes in part by restricting the vocabulary of alternatives to proven solutions. Advantages of this approach include enhanced reuse, more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability. Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design alternatives to a relatively small number.

Properties of software design follow from the choice of an architectural pattern. Those patterns that are more desirable for a particular problem should improve the implementation of the resulting design solution, perhaps by making it easier to arbitrate conflicting design constraints, by increasing insight into poorly understood design contexts, or by helping to surface inconsistencies in requirements. We will discuss architectural patterns in more detail in Chapter 13.

·­·­·- 0 A a

2.13. Providing a Basis for Training

The architecture, including a description of how the elements interact with each other to carry out the required behavior, can serve as the first introduction to the system for new project members. This reinforces our point that one of the important uses of software architecture is to support and encourage communication among the various stakeholders. The architecture is a common reference point.

Module views are excellent for showing someone the structure of a project: Who does what, which teams are assigned to which parts of the system, and so forth. Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

We will discuss these views in more detail in Chapter 18.

·­·­·- 0 A a

2.14. Summary

Software architecture is important for a wide variety of technical and nontechnical reasons. Our list includes the following:

1. An architecture will inhibit or enable a system's driving quality attributes.

2. The decisions made in an architecture allow you to reason about and manage change as the

system evolves.

3. The analysis of an architecture enables early prediction of a system's qualities.

4. A documented architecture enhances communication among stakeholders.

5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change

design decisions.

6. An architecture defines a set of constraints on subsequent implementation.

7. The architecture dictates the structure of an organization, or vice versa.

8. An architecture can provide the basis for evolutionary prototyping.

9. An architecture is the key artifact that allows the architect and project manager to reason

about cost and schedule.

10. An architecture can be created as a transferable, reusable model that forms the heart of a

product line.

11. Architecture-based development focuses attention on the assembly of components, rather

than simply on their creation.

12. An architecture channels the creativity of developers, reducing design and system

complexity.

13. An architecture can be the foundation for training of a new team member.

·­·­·- 0 A a

2.15. For Further Reading

Rebecca Grinter has observed architects from a sociological standpoint. In [Grinter 99] she argues eloquently that the architect's primary role is to facilitate stakeholder communication. The way she puts it is that architects enable communication among parties who would otherwise not be able to talk to each other.

The granddaddy of papers about architecture and organization is [Conway 68]. Conway's law states that "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations."

There is much about software development through composition that remains unresolved. When the components that are candidates for importation and reuse are distinct subsystems that have been built with conflicting architectural assumptions, unanticipated complications can increase the effort required to integrate their functions. David Garlan and his colleagues coined the term architectural mismatch to describe this situation, and their paper on it is worth reading [Garlan 95].

Paulish [Paulish 02] discusses architecture-based project management, and in particular the ways in which an architecture can help in the estimation of project cost and schedule.

·­·­·- 0 A a

2.16. Discussion Questions

1. For each of the thirteen reasons articulated in this chapter why architecture is important, take the

contrarian position: Propose a set of circumstances under which architecture is not necessary to achieve the result indicated. Justify your position. (Try to come up with different circumstances for each of the thirteen.)

2. This chapter argues that architecture brings a number of tangible benefits. How would you

measure the benefits, on a particular project, of each of the thirteen points?

3. Suppose you want to introduce architecture-centric practices to your organization. Your

management is open to the idea, but wants to know the ROI for doing so. How would you respond?

4. Prioritize the list of thirteen points in this chapter according to some criteria meaningful to you.

Justify your answer. Or, if you could choose only two or three of the reasons to promote the use of architecture in a project, which would you choose and why?

A a

·­·­·-

0

0

In 1976, a New Yorker magazine cover featured a cartoon by Saul Steinberg showing a New Yorker's view of the world. You've probably seen it; if not, you can easily find it online. Looking to the west from 9th Avenue in Manhattan, the illustration shows lOth Avenue, then the wide Hudson River, then a thin strip of cotnpletely nondescript land called "Jersey," followed by a somewhat thicker strip of land representing the entire rest of the United States. The mostly empty United States has a cartoon mountain or two here and there and a few cities haphazardly placed "out there," and is flanked by featureless "Canada" on the right and "Mexico" on the left. Beyond is the Pacific Ocean, only slightly wider than the Hudson, and beyond that lie tiny amorphous shapes for Japan and China and Russia, and that's pretty much the world from a New Yorker's perspective.

In a book about architecture, it is tempting to view architecture in the same way, as the most important piece of the software universe. And in some chapters, we unapologetically will do exactly that. But in this chapter we put software architecture in its place, showing how it supports and is informed by other critical forces and activities in the various contexts in which it plays a role.

These contexts, around which we structured this book, are as follows:

• Technical. What technical role does the software architecture play in the system or systems of which it's a part?

• Project life cycle. How does a software architecture relate to the other phases of a software development life cycle?

• Business. How does the presence of a software architecture affect an organization's business environment?

• Professional. What is the role of a software architect in an organization or a development project?

These contexts all play out throughout the book, but this chapter introduces each one. Although the contexts are unchanging, the specifics for your system may change over time. One challenge for the architect is to envision what in their context might change and to adopt mechanisms to protect the system and its development if the envisioned changes come to pass.

3. The Many Contexts of Software Architecture

People in London think of London as the center of the world, whereas New Yorkers think the world ends three miles outside of Manhattan.

-Toby Young

·­·­·- 0 A a

Architectures inhibit or enable the achievement of quality attributes, and one use of an architecture is to support reasoning about the consequences of change in the particular quality attributes important for a system at its inception.

Architectures Inhibit or Enable the Achievement of Quality Attributes

Chapter 2 listed thirteen reasons why software architecture is important and merits study. Several of those reasons deal with exigencies that go beyond the bounds of a particular development project (such as communication among stakeholders, many of whom may reside outside the project's organization). Others deal with nontechnical aspects of a project (such as the architecture's influence on a project's team structure, or its contribution to accurate budget and schedule estimation). The first three reasons in that List of Thirteen deal specifically with an architecture's technical impact on every system that uses it:

1. An architecture will inhibit or enable the achievement of a system's quality attributes.

2. You can predict many aspects of a system's qualities by studying its architecture.

3. An architecture makes it easier for you to reason about and manage change.

These are all about the architecture's effect on a system's quality attributes, although the first one states it the most explicitly. While all of the reasons enumerated in Chapter 2 are valid statements of the contribution of architecture, probably the most important reason that it warrants attention is its critical effect on quality attributes.

This is such a critical point that, with your indulgence, we'll add a few more points to the bullet list that we gave in Section 2.1. Remember? The one that started like this:

• If your system requires high performance, then you need to pay attention to managing the time­ based behavior of elements, their use of shared resources, and the frequency and volume of interelement communication.

To that list, we'll add the following:

• If you care about a system's availability, you have to be concerned with how components take over for each other in the event of a failure, and how the system responds to a fault.

• If you care about usability, you have to be concen1ed about isolating the details of the user interface and those elements responsible for the user experience from the rest of the system, so that those things can be tailored and improved over titne.

• If you care about the testability of your system, you have to be concerned about the testability of individual elements, which means making their state observable and controllable, plus understanding the emergent behavior of the elements working together.

• If you care about the safety of your system, you have to be concerned about the behavioral envelope of the elements and the emergent behavior of the elements working in concert.

• If you care about interoperability between your system and another, you have to be concerned about which elements are responsible for external interactions so that you can control those

3.1. Architecture in a Technical Context

·­·­·- 0 A a

These and other representations are all saying the same thing in different ways: If you care about this quality attribute, you have to be concerned with these decisions, all of which are thoroughly architectural in nature. An architecture inhibits or enables a system's quality attributes. And conversely, nothing else influences an architecture more than the quality attribute requirements it must satisfy.

If you care about architecture for no other reason, you should care about it for this one. We feel so strongly about architecture's importance with respect to achieving system quality attributes that all of Part II of this book is devoted to the topic.

Why is functionality missing frmn the preceding list? It is missing because the architecture mainly provides containers into which the architect places functionality. Functionality is not so much a driver for the architecture as it is a consequence of it. We return to this point in more detail in Part II.

Architectures and the Technical Environment

The technical environment that is current when an architecture is designed will influence that architecture. It might include standard industry practices or software engineering techniques prevalent in the architect's professional co1nmunity. It is a brave architect who, in today's environment, does not at least consider a web-based, object-oriented, service-oriented, mobility-aware, cloud-based, social­ networking-friendly design for an information system. It wasn't always so, and it won't be so ten years from now when another crop of technological trends has come to the fore.

The Swedish Ship Vasa

In the 1620s, Sweden and Poland were at war. The king of Sweden, Gustavus Adolphus, was determined to put a swift and favorable end to it and commissioned a new warship the likes of which had never been seen before. The Vasa, shown in Figure 3.1, was to be the world's most formidable instrument of war: 70 meters long, able to carry 300 soldiers, and with an astonishing 64 heavy guns mounted on two gun decks. Seeking to add overwhelming firepower to his navy to strike a decisive blow, the king insisted on stretching the Vasa's armaments to the limits. Her architect, Henrik Hybertsson, was a seasoned Dutch shipbuilder with an impeccable reputation, but the Vasa was beyond even his broad experience. Two-gun-deck ships were rare, and none had been built of the Vasa's size and armament.

interactions.

·­·­·- 0 A a

Figure 3.1. The warship. Used with permission of The Vasa Museum, Stockholm, Sweden.

Like all architects of systems that push the envelope of experience, Hybertsson had to balance many concerns. Swift time to deployment was critical, but so were performance, functionality, safety, reliability, and cost. He was also responsible to a variety of stakeholders. In this case, the primary customer was the king, but Hybertsson also was responsible to the crew that would sail his creation. Also like all architects, Hybertsson brought his experience with him to the task. In this case, his experience told him to design the Vasa as though it were a single-gun-deck ship and then extrapolate, which was in accordance with the technical environment of the day. Faced with an impossible task, Hybertsson had the good sense to die about a year before the ship was finished.

The project was completed to his specifications, however, and on Sunday morning, August 10, 1628, the mighty ship was ready. She set her sails, waddled out into Stockholm's deep-water harbor, fired her guns in salute, and promptly rolled over. Water poured in through the open gun ports, and the Vasa plummeted. A few minutes later her first and only voyage ended 30 meters beneath the surface. Dozens among her 150-man crew drowned.

Inquiries followed, which concluded that the ship was well built but "badly proportioned." In other words, its architecture was flawed. Today we know that Hybertsson did a poor job of balancing all of the conflicting constraints levied on him. In particular, he did a poor job of risk management and a poor job of customer managetnent (not that anyone could have fared better). He simply acquiesced in the face of impossible requirements.

The story of the Vasa, although more than 375 years old, well illustrates the Architecture Influence Cycle: organization goals beget requirements, which beget an architecture, which begets a system. The architecture flows from the architect's experience and the technical environment of the day. Hybertsson suffered from the fact

A ·­·­·- 0

a

that neither of those were up to the task before him.

In this book, we provide three things that Hybertsson could have used:

1. Examples of successful architectural practices that work under demanding

requirements, so as to help set the technical playing field of the day.

2. Methods to assess an architecture before any system is built from it, so as to

mitigate the risks associated with launching unprecedented designs.

3. Techniques for incremental architecture-based development, so as to uncover

design flaws before it is too late to correct them.

Our goal is to give architects another way out of their design dilemmas than the one that befell the ill-fated Dutch ship designer. Death before deployment is not nearly so admired these days.

-PCC

·­·­·- 0 A a

3.2. Architecture in a Project Life-Cycle Context

Software development processes are standard approaches for developing software systems. They impose a discipline on software engineers and, more important, teams of software engineers. They tell the members of the team what to do next. There are four dominant software development processes, which we describe in roughly the order in which they came to prominence:

1. Waterfall. For many years the Waterfall model dominated the field of software development.

The Waterfall model organized the life cycle into a series of connected sequential activities, each with entry and exit conditions and a formalized relationship with its upstream and downstream neighbors. The process began with requirements specification, followed by design, then implementation, then integration, then testing, then installation, all followed by maintenance. Feedback paths from later to earlier steps allowed for the revision of artifacts (requirements documents, design documents, etc.) on an as-needed basis, based on the knowledge acquired in the later stage. For example, designers might push back against overly stringent requirements, which would then be reworked and flow back down. Testing that uncovered defects would trigger reimplementation (and maybe even redesign). And then the cycle continued.

2. Iterative. Over time the feedback paths of the Waterfall model became so pronounced that it became clear that it was better to think of software development as a series of short cycles through the steps some requirements lead to some design, which can be implemented and tested while the next cycle's worth of requirements are being captured and designed. These cycles are called iterations, in the sense of iterating toward the ultimate software solution for the given problem. Each iteration should deliver something working and useful. The trick here is to uncover early those requirements that have the most far-reaching effect on the design; the corresponding danger is to overlook requirements that, when discovered later, will capsize the design decisions made so far. An especially well-known iterative process is called the Unified Process (originally named the Rational Unified Process, after Rational Software, which originated it). It defines four phases of each iteration: inception, elaboration, construction, and transition. A set of chosen use cases defines the goals for each iteration, and the iterations are ordered to address the greatest risks first.

3. Agile. The term "Agile software development" refers to a group of software developtnent

methodologies, the best known of which include Scrmn, Extreme Programming, and Crystal Clear. These methodologies are all incremental and iterative. As such, one can consider some iterative methodologies as Agile. What distinguishes Agile practices is early and frequent delivery of working software, close collaboration between developers and customers, self­ organizing teams, and a focus on adaptation to changing circumstances (such as late-arriving requirements). All Agile methodologies focus on teamwork, adaptability, and close collaboration (both within the team and between team members and customers/end users). These methodologies typically eschew substantial up-front work, on the assumption that requirements always change, and they continue to change throughout the project's life cycle. As such, it might seem that Agile methodologies and architecture cannot happily coexist. As

·­·­·- 0 A a

we will show in Chapter 15, this is not so.

4. Model-driven development. Model-driven development is based on the idea that humans

should not be writing code in programming languages, but they should be creating models of the domain, from which code is automatically generated. Humans create a platform­ independent model (PIM), which is combined with a platform-definition model (PDM) to generate running code. In this way the PIM is a pure realization of the functional requirements while the PDM addresses platform specifics and quality attributes.

All of these processes include design among their obligations, and because architecture is a special kind of design, architecture finds a home in each one. Changing from one development process to another in the middle of a project requires the architect to save useful information from the old process and determine how to integrate it into the new process.

No matter what software development process or life-cycle model you're using, there are a nutnber of activities that are involved in creating a software architecture, using that architecture to realize a complete design, and then implementing or managing the evolution of a target system or application. The process you use will determine how often and when you revisit and elaborate each of these activities. These activities include:

1. Making a business case for the system

2. Understanding the architecturally significant requirements

3. Creating or selecting the architecture

4. Documenting and communicating the architecture

5. Analyzing or evaluating the architecture

6. Implementing and testing the system based on the architecture

7. Ensuring that the implementation conforms to the architecture

Each of these activities is covered in a chapter in Part III of this book, and described briefly below.

Making a Business Case for the System

A business case is, briefly, a justification of an organizational investment. It is a tool that helps you make business decisions by predicting how they will affect your organization. Initially, the decision will be a go/no-go for pursuing a new business opportunity or approach. After initiation, the business case is reviewed to assess the accuracy of initial estimates and then updated to examine new or alternative angles on the opportunity. By documenting the expected costs, benefits, and risks, the business case serves as a repository of the business and marketing data. In this role, management uses the business case to determine possible courses of action.

Knowing the business goals for the system Chapter 16 will show you how to elicit and capture them in a systematic way is also critical in the creation of a business case for a system.

Creating a business case is broader than simply assessing the market need for a system. It is an important step in shaping and constraining any future requirements. How much should the product cost? What is its targeted market? What is its targeted time to market and lifetime? Will it need to interface with other systems? Are there system limitations that it must work within?

·­·­·- 0 A a

These are all questions about which the system's architects have specialized knowledge; they must contribute to the answers. These questions cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, the organization may be unable to achieve its business goals. Typically, a business case is created prior to the initiation of a project, but it also may be revisited during the course of the project for the organization to determine whether to continue making investments in the project. If the circumstances assumed in the initial version of the business case change, the architect may be called upon to establish how the system will change to reflect the new set of circumstances.

Understanding the Architecturally Significant Requirements

There are a variety of techniques for eliciting requirements from the stakeholders. For example, object­ oriented analysis uses use cases and scenarios to embody requirements. Safety-critical systems sometimes use tnore rigorous approaches, such as finite-state-machine models or formal specification languages. In Part II of this book, which covers quality attributes, we introduce a collection of quality attribute scenarios that aid in the brainstorming, discussion, and capture of quality attribute requirements for a system.

One fundamental decision with respect to the system being built is the extent to which it is a variation on other systems that have been constructed. Because it is a rare system these days that is not similar to other systems, requirements elicitation techniques involve understanding these prior systems' characteristics. We discuss the architectural implications of software product lines in Chapter 25.

Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model and explore desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly build support for the project and catalyze decisions on the system's design and the design of its user interface.

Creating or Selecting the Architecture

In the landmark book The Mythical Man-Month, Fred Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture. We firmly believe this as well. Good architecture almost never results as an emergent phenomenon.

Chapters 5-12 and 11 will provide practical techniques that will aid you in creating an architecture to achieve its behavioral and quality requirements.

Documenting and Communicating the Architecture

For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders. Developers must understand the work assignments that the architecture requires of them, testers must understand the task structure that the architecture imposes on them, management must understand the scheduling implications it contains, and so forth.

Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds. Architectural documentation should also be minimal and aimed at the stakeholders who will use it; we are no fans of documentation for documentation's

·­·­·- 0 A a

sake. We discuss the documentation of architectures and provide examples of good documentation practices in Chapter 18. We will also discuss keeping the architecture up to date when there is a change in something on which the architecture documentation depends.

Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these competing designs in a rational way is one of the architect's greatest challenges.

Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders' needs. Analysis techniques to evaluate the quality attributes that an architecture imparts to a system have become much more widespread in the past decade. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture. The most mature methodological approach is found in the Architecture Tradeoff Analysis Method (ATAM) of Chapter 21, while the economic implications of architectural decisions are explored in Chapter 23.

Implementing and Testing the System Based on the Architecture

If the architect designs and analyzes a beautiful, conceptually sound architecture which the implementers then ignore, what was the point? If architecture is important enough to devote the time and effort of your best minds to, then it is just as important to keep the developers faithful to the structures and interaction protocols constrained by the architecture. Having an explicit and well­ communicated architecture is the first step toward ensuring architectural conformance. Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture (as opposed to just the code) is better.

There are many reasons why developers might not be faithful to the architecture: It might not have been properly documented and disseminated. It might be too confusing. It might be that the architect has not built ground-level support for the architecture (particularly if it presents a different way of "doing business" than the developers are used to), and so the developers resist it. Or the developers may sincerely want to implement the architecture but, being human, they occasionally slip up. This is not to say that the architecture should not change, but it should not change purely on the basis of the whims of the developers, because they may not have the overall picture.

Ensuring That the Implementation Conforms to the Architecture

Finally, when an architecture is created and used, it goes into a maintenance phase. Vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase. And when they do get significantly out of sync, effort must be expended to either fix the implementation or update the architectural documentation.

Although work in this area is still relatively immature, it has been an area of intense activity in recent years. Chapter 20 will present the current state of recovering an architecture from an existing system and ensuring that it conforms to the specified architecture.

·­·­·- 0 A a

3.3. Architecture in a Business Context

Architectures and systems are not constructed frivolously. They serve some business purposes, although as mentioned before, these purposes may change over time.

Architectures and Business Goals

Systems are created to satisfy the business goals of one or more organizations. Development organizations want to make a profit, or capture market, or stay in business, or help their customers do their jobs better, or keep their staff gainfully employed, or make their stockholders happy, or a little bit of each. Customers have their own goals for acquiring a system, usually involving some aspect of making their lives easier or more productive. Other organizations involved in a project's life cycle, such as subcontractors or government regulatory agencies, have their own goals dealing with the system.

Architects need to understand who the vested organizations are and what their goals are. Many of these goals will have a profound influence on the architecture.

Many business goals will be manifested as quality attribute requirements. In fact, every quality attribute such as a user-visible response time or platform flexibility or ironclad security or any of a dozen other needs should originate from some higher purpose that can be described in terms of added value. If we ask, for example, "Why do you want this system to have a really fast response time?" we might hear that this will differentiate the product from its competition and let the developing organization capture market share.

Some business goals, however, will not show up in the form of requirements. We know of one software architect who was informed by his manager that the architecture should include a database. The architect was perplexed, because the requirements for the system really didn't warrant a database and the architect's design had nicely avoided putting one in, thereby simplifying the design and lowering the cost of the product. The architect was perplexed, that is, until the manager reminded the architect that the company's database department was currently overstaffed and underworked. They needed something to do! The architect put in the database, and all was well. That kind of business goal -keeping staff gainfully employed is not likely to show up in any requirements document, but if the architect had failed to meet it, the manager would have considered the architecture as unacceptable, just as the customer would have if it failed to provide a key piece of functionality.

Still other business goals have no effect on the architecture whatsoever. A business goal to lower costs might be realized by asking employees to work from home, or tum the office thermostats down in the winter, or using less paper in the printers. Chapter 16 will deal with uncovering business goals and the requirements they lead to.

Figure 3.2 illustrates the major points from the preceding discussion. In the figure, the arrows mean "leads to." The solid arrows highlight the relationships of most interest to us.

A a

·­·­·- 0

Figure 3.2. Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

Architectures and the Development Organization

A development organization contributes many of the business goals that influence an architecture. For example, if the organization has an abundance of experienced and idle programmers skilled in peer-to­ peer communications, then a peer-to-peer architecture might be the approach supported by management. If not, it may well be rejected. This would support the business goal, perhaps left implicit, of not wanting to hire new staff or lay off existing staff, or not wanting to invest significantly in the retraining of existing staff.

More generally, an organization often has an investment in assets, such as existing architectures and the products based on them. The foundation of a development project may be that the proposed system is the next in a sequence of similar systems, and the cost estimates assume a high degree of asset reuse and a high degree of skill and productivity from the programmers.

Additionally, an organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may view the proposed system as one means of financing and extending that infrastructure. For example, an organization may decide that it wants to develop a reputation for supporting solutions based on cloud computing or service-oriented architecture or high­ performance real-time computing. This long-term goal would be supported, in part, by infrastructural investments that will affect the developing organization: a cloud-computing group needs to be hired or grown, infrastructure needs to be purchased, or perhaps training needs to be planned.

Finally, the organizational structure can shape the software architecture, and vice versa. Organizations are often organized around technology and application concepts: a database group, a networking group, a business rules team, a user-interface group. So the explicit identification of a distinct subsystem in the architecture will frequently lead to the creation of a group with the name of the subsystem. Furthermore, if the user-interface team frequently needs to communicate with the business rules team, these teams will need to either be co-located or they will need some regular means of communicating and coordinating.

J I I I I v Non architectural Solutions

Quality Attributes

Architecture

·­·­·- 0 A a

3.4. Architecture in a Professional Context

What do architects do? How do you become an architect? In this section we talk about the many facets of being an architect that go beyond what you learned in a programming or software engineering course.

You probably know by now that architects need more than just technical skills. Architects need to explain to one stakeholder or another the chosen priorities of different properties, and why particular stakeholders are not having all of their expectations fulfilled. To be an effective architect, then, you will need diplomatic, negotiation, and communication skills.

You will perform many activities beyond directly producing an architecture. These activities, which we call duties, form the backbone of individual architecture competence. We surveyed the broad body of information aimed at architects (such as websites, courses, books, and position descriptions for architects), as well as practicing architects, and duties are but one aspect. Writers about architects also speak of skills and knowledge. For example, architects need the ability to communicate ideas clearly and need to have up-to-date knowledge about (for example) patterns, or database platforms, or web services standards.

Duties, skills, and knowledge form a triad on which architecture competence rests. You will need to be involved in supporting management and dealing with customers. You will need to manage a diverse workload and be able to switch contexts frequently. You will need to know business considerations. You will need to be a leader in the eyes of developers and management. In Chapter 24 we examine at length the architectural competence of organizations and people.

Architects' Background and Experience

We are all products of our experiences, architects included. If you have had good results using a particular architectural approach, such as three-tier client-server or publish-subscribe, chances are that you will try that same approach on a new development effort. Conversely, if your experience with an approach was disastrous, you may be reluctant to try it again.

Architectural choices may also come from your education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well. You may also wish to experiment with an architectural pattern or technique learned from a book (such as this one) or a training course.

Why do we mention this? Because you (and your organization) must be aware of this influence, so that you can manage it to the best of your abilities. This may mean that you will critically examine proposed architectural solutions, to ensure that they are not simply the path of least resistance. It may mean that you will take training courses in interesting new technologies. It may mean that you will invest in exploratory projects, to "test the water" of a new technology. Each of these steps is a way to proactively manage your background and experience.

·­·­·- 0 A a

3.5. Stakeholders

Many people and organizations are interested in a software system. We call these entities stakeholders. A stakeholder is anyone who has a stake in the success of the system: the customer, the end users, the developers, the project manager, the maintainers, and even those who market the system, for example. But stakeholders, despite all having a shared stake in the success of the system, typically have different specific concerns that they wish the system to guarantee or optimize. These concerns are as diverse as providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving short time to market or low cost of development, gainfully employing programmers who have a particular specialty, or providing a broad range of functions. Figure 3.3 shows the architect receiving a few helpful stakeholder "suggestions."

Neat features, short time to market, low co.st, parity with competing products!

Low cost, keeping people emptoyed!

Modifiability!

Behavior, performance, security, rel.iability,

Low cost, timely usability!

delivery, not changed very often!

Figure 3.3. Influence of stakeholders on the architect

You will need to know and understand the nature, source, and priority of constraints on the project as early as possible. Therefore, you must identify and actively engage the stakeholders to solicit their needs and expectations. Early engagement of stakeholders allows you to understand the constraints of

·­·­·- 0 A a

the task, manage expectations, negotiate priorities, and make tradeoffs. Architecture evaluation (covered in Part III of this book) and iterative prototyping are two means for you to achieve stakeholder engagement.

Having an acceptable system involves appropriate performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior. All of these qualities, and others, affect how the delivered system is viewed by its eventual recipients, and so such quality attributes will be demanded by one or more of the system's stakeholders.

The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory. It is a rare requirements document that does a good job of capturing all of a system's quality requirements in testable detail (a property is testable if it is falsifiable; "make the system easy to use" is not falsifiable but "deliver audio packets with no more than 10 ms. jitter" is falsifiable). The architect often has to fill in the blanks the quality attribute requirements that have not been explicitly stated and mediate the conflicts that frequently emerge.

Therefore, one of the best pieces of advice we can give to architects is this: Know your stakeholders. Talk to them, engage them, listen to them, and put yourself in their shoes. Table 3.1 enumerates a set of stakeholders. Notice the remarkable variety and length of this set, but remember that not every stakeholder named in this list may play a role in every system, and one person may play many roles.

Name

Analyst

Architect

Business Manager

Conformance Checker

Customer

Database Administrator

Oeptoyer

Designer

Table 3.1. Stakeholders for a System and Their Interests

Description

Responsible for analyzu'lg the architecture to make sure it meets certain critical quality attribute requirements. Analysts are often specialized; for instance, performance analysts. safety analysts, and security analysts may have well·defined positions in a project.

Responsible for the development of the architecture and its documentation. Focus and responsibility is on the system.

Responsible for the functioning of the business/organizational en1ily thai owns the system. Includes managerial/executive responsibility, responsibility for defining business processes, etc.

Aesponstble for assunng conformance to standards and processes to provide confidence in a product's suitability.

Pays for the system and ensures itsdelivery. The customer often speaks for or represents the end user. especially In a government acquisition context.

Involved fn many aspects of the data stores, Including database design,

data analysis. data modeling and optimization, installation of database software, and monitoring and administration of database security.

Responsible for accepting the completed system from the development effort and deploying it, making it operational. and fulfilling its allocated business function.

Responsible for systems and/or software design downstream or the archilecture, applying the archilecture to meet specific requirements of the parts for which they are responsible.

Interest in Architecture

Interest in Architecture

Analyzing satisfaction of quality attlibute requirements of the system based on its architecture.

Analyzing satisfaction of quality attlibute requirements of the system based on its architecture.

Negotiating and making tradeofls among competing requirements and design approaches. A vessel for recording design decisions. Providing evidencethat the architecture satisfies its requirements.

Negotiating and making tradeofls among competing requirements and design approaches. A vessel for recording design decisions. Providing evidencethat the architecture satisfies its requirements.

Understanding the ability of the architecture to meet business goals.

Understanding the ability of the architecture to meet business goals.

Basis for conformancechecking, for assurance that Implementations have been fa�hful to the architectural prescriptions.

Basis for conformancechecking, for assurance that Implementations have been fa�hful to the architectural prescriptions.

Assuring required functionality and quality will be delivered; gauging progress; estimating cost; and setting expectations for What will be delivered, when, and for how much.

Assuring required functionality and quality will be delivered; gauging progress; estimating cost; and setting expectations for What will be delivered, when, and for how much.

Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.

Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.

Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.

Understanding the architectural elements that are delivered and to be installed at the customer or end user's site, and their overall responsibility toward system function.

Understanding the architectural elements that are delivered and to be installed at the customer or end user's site, and their overall responsibility toward system function.

Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understand­ ing how their part will communicate and interact with other parts of the system.

Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understand­ ing how their part will communicate and interact with other parts of the system.

Evaluator

Implementer

Integrator

Maintainer

Network Administrator

Product-line Manager

Project Manager

Representative of External Systems

System Engineer

Tester

User

Responsible for conducting a formal evaluation of the architecture (and hs documentation) against some clearly defined criteria.

Responsible for the development of specific elements according to designs, requirements. and the architecture.

Responsible for taking individual components and integrating them, according to the architecture and system designs.

Responsible lor fixing bugs and providing enhancements to the system throughout its life (Including adaptation of the system lor uses not originally envisioned).

Responsible lor the maintenance and oversight of computer hardware and software in a computer network. This may include the deployment. configuration. maintenance, and monitoring ol network components.

Responsible tor development of an entire family of products, all built using the same core assets (including the architecture).

Responsible for planning. sequencing, scheduling, and allocating resources to develop software components and deliver components to integration and test activities.

Responsible for managing a system with which this one must interoperate, and its interface with our system.

Responsible for design and development of systems or system components in which software plays a role.

Responsible for the (independent) test and verification of the system or its elements against the formal requirements and the arcMecture.

The actual end users of the system. There may bedistinguished kinds of users. such as administrators. superusers, etc.

·­·­·- 0

Evaluating the architecture's abilhy to deliver required behavior and quality attributes.

Evaluating the architecture's abilhy to deliver required behavior and quality attributes.

Understanding inviolable constraints and exploitable freedoms on development activities.

Understanding inviolable constraints and exploitable freedoms on development activities.

Producing integration plans and procedures, and locating the source of integration !allures.

Producing integration plans and procedures, and locating the source of integration !allures.

Understanding the ramifications of a change.

Understanding the ramifications of a change.

Determining network loads during various useprofiles, understanding uses of the network.

Determining network loads during various useprofiles, understanding uses of the network.

Determining whether a potential new member of a product family is in or out of scope and, if out. by how much.

Determining whether a potential new member of a product family is in or out of scope and, if out. by how much.

Helping to set budget and schedule. gauging progress against established budget and schedule, identifying and resolving development-time resource contention.

Helping to set budget and schedule. gauging progress against established budget and schedule, identifying and resolving development-time resource contention.

Defining the set of agreement between the systems.

Defining the set of agreement between the systems.

Assuring that the system environment provided for the software is sufficient.

Assuring that the system environment provided for the software is sufficient.

Creating tests based on the behavior and interaction of the software elements.

Creating tests based on the behavior and interaction of the software elements.

Users, in the role of reviewers, might use architecture documentation to check whether desired functionality is being delivered. Users might also use the documentation to understand what the major system elements are, which can aid them In emergency field ma1ntenance.

Users, in the role of reviewers, might use architecture documentation to check whether desired functionality is being delivered. Users might also use the documentation to understand what the major system elements are, which can aid them In emergency field ma1ntenance.

A a

·­·­·- 0 A a

3.6. How Is Architecture Influenced?

For decades, software designers have been taught to build systems based on the software's technical requirements. In the older Waterfall model, the requirements document is "tossed over the wall" into the designer's cubicle, and the designer must come forth with a satisfactory design. Requirements beget design, which begets system. In an iterative or Agile approach to development, an increment of requirements begets an increment of design, and so forth.

This vision of software development is short-sighted. In any development effort, the requirements make explicit some but only some of the desired properties of the final system. Not all requirements are focused directly on desired system properties; some requirements might mandate a development process or the use of a particular tool. Furthermore, the requirements specification only begins to tell the story. Failure to satisfy other constraints may render the system just as problematic as if it functioned poorly.

What do you suppose would happen if two different architects, working in two different organizations, were given the same requirements specification for a system? Do you think they would produce the same architecture or different ones?

The answer is that they would very likely produce different ones, which immediately belies the notion that requirements determine architecture. Other factors are at work.

A software architecture is a result of business and social influences, as well as technical ones. The existence of an architecture in tum affects the technical, business, and social environments that subsequently influence future architectures. In particular, each of the contexts for architecture that we just covered technical, project, business, and professional plays a role in influencing an architect and the architecture, as shown in Figure 3.4.

Architecfs Influences

Busjness

Technical ... .... Stakeholders ... ,

Project

Professional Architect

Figure 3.4. Influences on the architect

An architect designing a syste1n for which the real-time deadlines are tight will make one set of design choices; the same architect, designing a similar system in which the deadlines can be easily satisfied, will make different choices. And the same architect, designing a non-real-time system, is likely to make quite different choices still. Even with the same requirements, hardware, support

·­·­·- 0 A a

software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.

·­·­·- 0 A a

3.7. What Do Architectures Influence?

The story about contexts influencing architectures has a flip side. It turns out that architectures have an influence on the very factors that influence them. Specifically, the existence of an architecture affects the technical, project, business, and professional contexts that subsequently influence future architectures.

Here is how the cycle works:

• Technical context. The architecture can affect stakeholder requirements for the next system by giving the customer the opportunity to receive a system (based on the same architecture) in a more reliable, timely, and economical manner than if the subsequent system were to be built from scratch, and typically with fewer defects. A customer may in fact be willing to relax some of their requirements to gain these economies. Shrink-wrapped software has clearly affected people's requirements by providing solutions that are not tailored to any individual's precise needs but are instead inexpensive and (in the best of all possible worlds) of high quality. Software product lines have the same effect on customers who cannot be so flexible with their requirements.

• Project context. The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system; as we will see, it particularly prescribes the units of software that must be implemented (or otherwise obtained) and integrated to form the system. These units are the basis for the development project's structure. Teams are formed for individual software units; and the development, test, and integration activities all revolve around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. If a company becomes adept at building families of similar systems, it will tend to invest in each team by nurturing each area of expertise. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization. In any design undertaken by the organization at large, these groups have a strong voice in the system's decomposition, pressuring for the continued existence of the portions they control.

• Business context. The architecture can affect the business goals of the developing organization. A successful system built from an architecture can enable a company to establish a foothold in a particular market segment think of the iPhone or Android app platforms as examples. The architecture can provide opportunities for the efficient production and deployment of similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.

• Professional context. The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base. A system that was successfully built around a particular technical approach will make the architect more inclined to build systems using the same approach in the future. On the other hand, architectures that fail are less likely to be chosen for future projects.

These and other feedback mechanisms form what we call the Architecture Influence Cycle, or AIC,

·­·­·- 0 A a

illustrated in Figure 3.5, which depicts the influences of the culture and business of the development organization on the software architecture. That architecture is, in turn, a primary determinant of the properties of the developed system or systems. But the AIC is also based on a recognition that shrewd organizations can take advantage of the organizational and experiential effects of developing an architecture and can use those effects to position their business strategically for future projects.

Architect's Influences

Business ----.

Technica�l---+-�Stakeholders 1-

P roject---.J

Professional ---� Architect

Figure 3.5. Architecture Influence Cycle

·­·­·- 0 A a

3.8. Summary

Architectures exist in four different contexts.

1. Technical. The technical context includes the achievement of quality attribute requirements.

We spend Part II discussing how to do this. The technical context also includes the current technology. The cloud (discussed in Chapter 26) and mobile computing (discussed in Chapter 27) are important current technologies.

2. Project life cycle. Regardless of the software development methodology you use, you must

make a business case for the system, understand the architecturally significant requirements, create or select the architecture, document and communicate the architecture, analyze or evaluate the architecture, implement and test the system based on the architecture, and ensure that the implementation conforms to the architecture.

3. Business. The system created from the architecture must satisfy the business goals of a wide

variety of stakeholders, each of whmn has different expectations for the system. The architecture is also influenced by and influences the structure of the development organization.

4. Professional. You must have certain skills and knowledge to be an architect, and there are

certain duties that you must perform as an architect. These are influenced not only by coursework and reading but also by your experiences.

An architecture has some influences that lead to its creation, and its existence has an impact on the architect, the organization, and, potentially, the industry. We call this cycle the Architecture Influence Cycle.

·­·­·- 0 A a

3.9. For Further Reading

The product line framework produced by the Software Engineering Institute includes a discussion of business cases from which we drew [SEI 12].

The SEI has also published a case study of Celsius Tech that includes an example of how organizations and customers change over time (Brownsword 96].

Several other SEI reports discuss how to find business goals and the business goals that have been articulated by certain organizations [Kazman 05, Clements 1 Ob].

Ruth Malan and Dana Bredemeyer provide a description of how an architect can build buy-in within an organization [Malan 00].

·­·­·- 0 A a

3.1 0. Discussion Questions

1. Enumerate six different software systems used by your organization. For each of these systems:

a. What are the contextual influences?

b. Who are the stakeholders?

c. How do these systems reflect or impact the organizational structure?

2. What kinds of business goals have driven the construction of the following:

a. The World Wide Web

b. Amazon's EC2 cloud infrastructure

c. Google's Android platform

3. What mechanisms are available to improve your skills and knowledge? What skills are you

lacking?

4. Describe a system you are familiar with and place it into the AIC. Specifically, identify the

forward and reverse influences on contextual factors.

A a

·­·­·- 0

In Part II, we provide the technical foundations for you to design or analyze an architecture to achieve particular quality attributes. We do not discuss design or analysis processes here; we cover those topics in Part III. It is impossible, however, to understand how to improve the performance of a design, for example, without understanding something about performance.

In Chapter 4 we describe how to specify a quality attribute requirement and motivate design techniques called tactics to enable you to achieve a particular quality attribute requirement. We also enumerate seven categories of design decisions. These are categories of decisions that are universally important, and so we provide material to help an architect focus on these decisions. In Chapter 4, we describe these categories, and in each of the following chapters devoted to a particular quality attribute

Chapters 5 1 1 we use those categories to develop a checklist that tells you how to focus your attention on the important aspects associated with that quality attribute. Many of the items in our checklists may seem obvious, but the purpose of a checklist is to help ensure the completeness of your design and analysis process.

In addition to providing a treatment of seven specific quality attributes (availability, interoperability, modifiability, performance, security, testability, and usability), we also describe how you can generate the material provided in Chapters 5-1 1 for other quality attributes that we have not covered.

Architectural patterns provide known solutions to a number of common problems in design. In Chapter 13, we present some of the most important patterns and discuss the relationship between patterns and tactics.

Being able to analyze a design for a particular quality attribute is a key skill that you as an architect will need to acquire. In Chapter 14, we discuss modeling techniques for some of the quality attributes.

Part Two. Quality Attributes

A a

4. Understanding Quality Attributes

·­·­·- 0

As we have seen in the Architecture Influence Cycle (in Chapter 3), many factors determine the qualities that must be provided for in a system's architecture. These qualities go beyond functionality, which is the basic statement of the system's capabilities, services, and behavior. Although functionality and other qualities are closely related, as you will see, functionality often takes the front seat in the development scheme. This preference is shortsighted, however. Systems are frequently redesigned not because they are functionally deficient the replacements are often functionally identical but because they are difficult to maintain, port, or scale; or they are too slow; or they have been compromised by hackers. In Chapter 2, we said that architecture was the first place in software creation in which quality requirements could be addressed. It is the mapping of a system's functionality onto software structures that determines the architecture's support for qualities. In Chapters 5-1 1 we discuss how various qualities are supported by architectural design decisions. In Chapter 17 we show how to integrate all of the quality attribute decisions into a single design.

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

In this chapter our focus is on understanding the following:

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

• How to achieve those qualities

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

This chapter provides the context for the discussion of specific quality attributes in Chapters 5-1 1.

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

·­·­·- 0 A a

4.1. Architecture and Requirements

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

1. Functional requirements. These requirements state what the system must do, and how it must

behave or react to runtime stimuli.

2. Quality attribute requirements. These requirements are qualifications of the functional

requirements or of the overall product. A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input. A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.

3. Constraints. A constraint is a design decision with zero degrees of freedom. That is, it's a

design decision that's already been 1nade. Examples include the requirement to use a certain programming language or to reuse a certain existing module, or a managetnent fiat to make your system service oriented. These choices are arguably in the purview of the architect, but exten1al factors (such as not being able to train the staff in a new language, or having a business agreement with a software supplier, or pushing business goals of service interoperability) have led those in power to dictate these design outcomes.

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

1. Functional requirements are satisfied by assigning an appropriate sequence of responsibilities

throughout the design. As we will see later in this chapter, assigning responsibilities to architectural elements is a fundamental architectural design decision.

2. Quality attribute requirements are satisfied by the various structures designed into the

architecture, and the behaviors and interactions of the elements that populate those structures. Chapter 17 will show this approach in more detail.

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

affected design decisions.