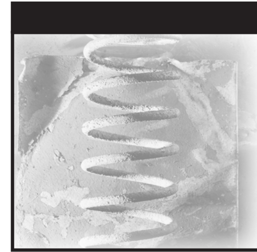
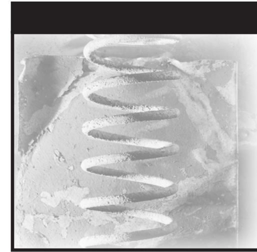


The Art of Software Architecture

Design Methods and Techniques

Stephen T. Albin

Wiley Publishing, Inc.



The Art of Software Architecture

Design Methods and Techniques

Stephen T. Albin

Wiley Publishing, Inc.

Executive Publisher: Joe Wikert
Executive Editor: Robert M. Elliott
Assistant Developmental Editor: Emilie Herman
Editorial Manager: Kathryn A. Malm
Assistant Managing Editor: Vincent Kunkemueller
Text Design & Composition: Wiley Composition Services

This book is printed on acid-free paper. ♾
Copyright © 2003 by Stephen T. Albin. All rights reserved.
Published by Wiley Publishing, Inc., Indianapolis, Indiana
Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8700. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: permcoordinator@wiley.com.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley Publishing logo and related trade dress are trademarks or registered trademarks of Wiley Publishing, Inc., in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

Albin, Stephen, 1967-
The art of software architecture : design methods and techniques / Stephen T. Albin.
p. cm.
Includes bibliographical references and index.
ISBN 0-471-22886-9
1. Computer software—Development.
2. Computer architecture. I. Title.
QA76.76.D47 A398 2003
005.1—dc21
2002155539

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To Jessie, Morgan, and Hannah
for their love and inspiration.

Contents

Acknowledgments		**xiii**
About the Author		**xv**
Introduction		**xvii**
Chapter 1	**Introduction to Software Architecture**	**1**
	Evolution of Software Development	2
	Fundamentals of Software Engineering	5
	Reusable Assets	6
	General-Purpose Programming Languages	7
	Special-Purpose Programming Languages	8
	Modeling Languages and Notations	8
	Elements of Software Architecture	9
	Components, Connectors, and Qualities	9
	Architectural Description	12
	Software Architecture versus Software Design Methodologies	13
	Types of Architecture	14
	Summary	16
Chapter 2	**The Software Product Life Cycle**	**17**
	Management View	18
	Inception Phase	20
	Elaboration Phase	20
	Construction Phase	21
	Transition Phase	21
	Software Engineering View	21
	Requirements Analysis and Specification	24
	Design	25
	Implementation and Testing	25
	Deployment and Maintenance	26

Engineering Design View 26
Product Planning: Specification of Information 29
Conceptual Design: Specification of Principle 29
Embodiment Design: Specification of Layout 30
Detail Design: Specification of Production 30
Architectural View 31
Predesign Phase 32
Domain Analysis Phase 33
Schematic Design Phase 34
Design Development Phase 34
Building Phases 34
Synthesizing the Views 35
Summary 37

Chapter 3 The Architecture Design Process 39
Understanding the Problem 41
Identifying Design Elements and Their Relationships 42
Defining System Context 47
Identifying Modules 48
Describing Components and Connectors 50
Evaluating the Architecture 51
Transforming the Architecture 52
Summary 53

Chapter 4 Introduction to Software Design 55
Problems in Software Architectural Design 56
Function, Form, and Fabrication: The Vitruvian Triad 57
Function and Product Planning 58
Form and Interaction Design 59
Cognitive Friction and Architectural Design 60
Fabrication 61
Application Architecture 62
Example 62
The Scope of Design 65
Tasks and Activities of Design 65
Origin of the Task 66
Organization 67
Novelty 68
Production 69
Technology 69
Horizontal Domain 70
Quality Attributes 70
Architecture versus Engineering Design 71
The Psychology and Philosophy of Design 72
Problems, Obstacles, and Solutions 72
Aristotelian Reasoning 73

General Methodology of Design 75
Purposeful Thinking 76
Analysis 77
Abstraction 78
Synthesis 78
General Heuristics 79
The Method of Persistent Questions 79
The Method of Negation 79
The Method of Forward Steps 80
The Method of Backward Steps 80
The Method of Factorization 81
The Method of Systematic Variation 81
Division of Labor and Collaboration 81
Summary 82

Chapter 5 Complexity and Modularity 85
Complexity 89
Understanding Complexity 89
Granularity and Context 90
Modularity 95
Architecture and Modules 96
Importing and Exporting 96
Coupling and Cohesion 97
Design Elements and Design Rules 98
Task Structure Matrix 103
Modular Operators 104
Splitting 106
Substituting 109
Augmenting and Excluding 109
Inversion 110
Porting 110
Summary 111

Chapter 6 Models and Knowledge Representation 113
What Are Models? 114
The Language of Models 115
Models and Human Comprehension 117
What Are Models Used For? 117
Systems Analysis Models 119
Systems Inference Models 119
Systems Design Models 120
What Roles Do Models Play? 120
Communication between Stakeholders and the Architect 121
Design Decisions and Design Assessment 121
Guidelines for Detail Design 122
Reusable Technical Artifacts 122

Modeling the Problem and Solution Domains 122
Problem Domain Models 123
Understanding the Problem 123
Analyzing the Requirements 123
Solution Domain Models 123
Technology-Independent Models 124
Technology-Dependent Models 124
Views 124
Objectives and Purpose Models 126
Behavioral/Functional Models 127
User Interface Prototypes 128
Scenarios and Threads 128
State Transition Diagrams 129
Process Models 129
Information/Data Models 130
Models of Form 131
Scale Models 131
Components and Connectors 132
Source Code 133
Nonfunctional/Performance Models 133
Summary 133

Chapter 7 Architecture Representation 135
Goals of Architecture Representation 136
Foundations of Software Architecture Representation 137
Fundamental Software Design Views 139
Architecture Description Languages 140
Design Language Elements 141
Composition 143
Abstraction 144
Reusability 144
Configuration 145
Heterogeneity 145
Architecture Analysis 145
First-Class Connectors 146
Modules and Components 146
Example: C2 SADL 148
Applying ADLs 149
Summary 150

Chapter 8 Quality Models and Quality Attributes 151
Process and Product Quality 153
Specifying Quality Requirements 153
Measuring Quality Attributes 154
Quality Requirements and Architectural Design 155
Systems Knowledge and Quality Attributes 156
Barriers to Achieving Quality 156

Common Quality Attribute Misunderstandings 157
Specifying Quality Requirements 157
Modeling Methods Don't Address Quality Attributes 157
Designing for Quality Attributes 157
Solution Catalogues and Quality Attributes 158
Quality Control Is an Afterthought 158
Understanding Quality Models 159
Benefits of Quality Models 166
Architecting with Quality Attributes 167
Functionality 167
Interoperability 168
Security 168
Performance (Efficiency) 168
Resource Efficiency 169
Modifiability 170
Availability and Reliability 170
Recoverability 170
Usability 171
Portability 171
Architecting and Quality Models 173
Summary 174

Chapter 9 Architectural Design Principles 175
Architectural Level of Design 176
Applying Design Principles 176
Using Systems Thinking 177
Example 178
Architecting with Design Operators 179
Decomposition 180
Identifying Functional Components 181
Composition/Aggregation 182
Component Communication 182
Replication 182
Compression 184
Abstraction 185
Virtual Machines and Adaptability 185
Resource Sharing 186
Functional Design Strategies 187
Self-Monitoring 187
Recovery 188
Instrumenting 188
Summary 188

Chapter 10 Applying Architectural Styles and Patterns 189
Defining Architectural Patterns and Style 190
Activation Model 192
Styles and Quality Attributes 195

Common Architectural Styles 196
Dataflow Systems 197
Call-and-Return Systems 200
Independent Components 202
Virtual Machines 203
Repositories 204
Example of Applying Architectural Styles 204
Summary 211

Chapter 11 Understanding Metamodels 213
Understanding Metamodels 214
Three-Layer Model of Knowledge Representation 215
Applying Reference Models 219
Seeheim Model 220
Arch/Slinky Model 223
Enterprise Application Reference Model 225
Technology Stacks and Architectural Layers 227
Fundamental Metamodel for Describing Software Components 230
Examples: Content Management System Reference Models 231
Domain Model 232
Content Collaboration Reference Model 234
Content Management Reference Model 236
Summary 237

Chapter 12 Creating Architectural Descriptions 239
Standardizing Architectural Descriptions 240
Creating an Architectural Description 241
Identify the Architectural Description 242
Identify Stakeholders 243
Select Viewpoints 244
Specify Viewpoints 245
Viewpoint Rationale 246
Viewpoints and Systems Knowledge 247
Interdependence of Views 247
Traceability 248
Methodologies and Viewpoints 248
Specify Views 250
Record View Inconsistencies 250
Create Architectural Rationale 251
Applying the Architectural Description 251
Creating an Architectural Description for an Existing System 252
Performing an Architectural Assessment 253
Specification Pragmatics 253
Summary 254

Chapter 13 Using Architecture Frameworks 255
Software Architecture Frameworks 256
Philosophies of Architecture Frameworks 257

Architecture Framework Goals 258
Methodologies and Architecture Frameworks 258
The 4+1 View Model of Architecture 259
Relationship to IEEE 1471 260
Logical Viewpoint 260
Stakeholders and Concerns Addressed 261
View Construction 261
Process Viewpoint 261
Stakeholders and Concerns Addressed 261
View Construction 262
Development Viewpoint 263
Stakeholders and Concerns Addressed 263
View Construction 263
Physical Viewpoint 263
Stakeholders and Concerns Addressed 263
View Construction 264
Scenario Viewpoint 264
Stakeholders and Concerns Addressed 264
View Construction 264
Model Overloading 264
Architecting with the Unified Process 265
Reference Model for Open Distributed Processing 266
Enterprise Viewpoint 267
Stakeholders and Concerns Addressed 267
View Construction 267
Information Viewpoint 268
Stakeholders and Concerns Addressed 269
View Construction 269
Computational Viewpoint 269
Stakeholders and Concerns Addressed 270
View Construction 270
Engineering Viewpoint 270
Stakeholders and Concerns Addressed 271
View Construction 271
Technology Viewpoint 271
Stakeholders and Concerns Addressed 271
View Construction 272
Summary 272

Chapter 14 Software Architecture Quality 273
Importance of Assessing Software Architecture 275
Content Publishing System Example 275
Enterprise Application Example 277
How to Improve Quality 277
Systematic Design Process 278
Understand the Right Problem 279
System Level View of Requirements 280
Differentiating Design and Requirements 281

Assessing Software Architectures 281
Scenarios: Reifying Nonfunctional Requirements 283
The Role of the Architectural Description 285
Architecture Evaluation 285
Assessing Modifiability 287
Assessing Performance 291
Summary 294

Appendix: Bibliography 297

Index 301

Acknowledgments

I would like to thank Scott Seaton, Steve Richard, and Stuart Thompto at ListenPoint for allowing me the time to complete this project.

I would also like to thank the staff at Wiley Publishing, Inc., and especially Emilie Herman for her excellent assistance reviewing and revising the manuscript.

Finally, I wish to thank my wife, Jessie, for her patience and support, and my daughters, Morgan and Hannah, for sharing me with this project.

About the Author

Stephen T. Albin is a software engineer and consultant in northern California and has developed commercial enterprise software applications, platforms, and technologies. He is a member of the ACM and IEEE Computer and Engineering Management Societies. He can be reached at stevealbin@computer.org.

Introduction

Software architecture is often confused with low-level design and the technology stack. Technology vendors and popular technology-focused journals tend to propagate this misunderstanding. As a result, many software engineers produce architecture descriptions that are nothing more than regurgitated diagrams of technology layers. The classic enterprise application architecture is often a diagram of so-called architectural layers depicting a presentation layer on top of a business logic layer (or middle-tier) on top of a persistence layer. This representation communicates nothing about how the system handles the functional or nonfunctional requirements of the system. It merely shows the technology to be used and how that technology will be integrated.

There is a temptation to assume that the layers of an application architecture map directly to individual technologies: Presentation is composed of Java Servlets and Java Server Pages (JSP); the business layer is composed of Enterprise JavaBeans (EJB); and the persistence layer is a relational database management system (RDBMS). For some simple systems, there is a one-to-one correspondence between the architectural layers and individual technologies. Those assumptions quickly become fallacies when the system becomes functionally more complex. Presentation logic may be composed not only of the servlets and JSP but also of EJBs and data stored in a relational database (for example, user preferences). Business logic may be composed not only of middle-tier EJB objects but also stored procedures and database triggers as well as other component technologies such as business rules engines and workflow engines.

For one system that I had to redesign, the only architectural description that existed was just such a technology stack. It depicted how eXtensible Markup Language (XML) documents would be passed between Java Servlets and

Enterprise JavaBeans as a flexible approach to creating the middle-tier application programming interface (API). It said nothing about how the system was composed of a main business logic subsystem, a security subsystem, and a reporting subsystem. Instead, it focused on how XML documents would be mapped to and from the relational database tables. Engineers on the project would often draw whiteboard diagrams of the system to include these three subsystems, as well as several other functional modules. The reality was that these were not modules. There was no separation or decoupling between any of them. The reporting module was composed of some user interface code that queried data from the same database tables that the other functions of the system operated on. The security logic was just an aspect of the system. There was no discernable security module; instead, the security logic was embedded in many objects throughout the system. The development organization structured itself around the presentation layer and everything else, treating the user interface as if it were a true module. The resulting system was difficult and costly to develop and maintain.

Software architecting involves the design of a system from multiple viewpoints. The common viewpoints used in software engineering are the technology stack (or physical) view, the object (or data) model, and the use case (or behavioral) view. These viewpoints are useful and necessary because they capture many types of design decisions and represent many system qualities such as functionality, information, and physical construction. They do not represent many other important system quality attributes such as modifiability, buildability, security, reliability, and performance, nor do they represent non-operational or business-oriented qualities such as the ability to reduce development and maintenance costs.

The problem with representing an architecture with this single technology-focused view is that we only see a vertical slice through a multidimensional system. Many architectural decisions cannot be represented in this view. If this is the only view we create, then we will probably neglect the other views to the detriment of the system itself.

An often ignored architectural viewpoint is the component or subsystem view of a system. By definition, a system is an aggregation of cooperating components. Without this view the system appears as a single module, despite the fact that engineers may talk about the security subsystem or the reporting system. It's easy to draw a few boxes and arrows on a whiteboard, but if these boxes and arrows don't mean anything, then we shouldn't bother.

A module has a clear interface that other modules import. The internals of the module are free to change. A Java Database Connectivity (JDBC) driver is an example. Applications rely on the published JDBC Java interfaces. Many vendors produce implementations, but they all conform to the same interface and therefore can be replaced. If an element of code cannot be replaced by another implementation without causing other elements to change, then that element of

code is not a module. What makes a system modular is the relatively small amount of information shared between the modules and the development teams designing and implementing those modules. Treating something as if it were a module will only frustrate the developers and managers.

In the above system, one of the first true decompositions of the system was the separation of the reporting system from the operational (or transactional) system. The results were tremendous, especially given the simplicity of the decomposition. No longer were there performance problems with running queries against the operational tables. No longer were operational and reporting use cases intertwined. The system was much easier to develop and maintain. In hindsight, the separation of the system into these two modules seems obvious and trivial. Yet when no one was looking at the system from this point of view, it was far from obvious, and the problems incurred were great. This little design decision can even be expressed as a software architecture pattern: *Separate operational data from analytic data* so that the two are loosely coupled such that they may be designed, developed, and maintained fairly independently and so that the system may have better performance.

Software architecture is emerging as a new discipline in software development in response to the growing complexity of software systems and the problems they are attempting to solve. Software is becoming the dominant component of many systems, and it is necessary for the community to develop new practices, principles, and standards so that we may somehow manage the growing complexity.

There are a couple of philosophies concerning how to improve the software crisis. One approach is to improve the quality of the software development process. In this school of thought, quality can be improved by using iterative development techniques, rapid application development (RAD) tools, frequent integration and testing, and keeping careful records so that an organization can build up historical data that will aid in improving the process in future product cycles. It uses iterative/increment development processes like the Rational Unified Process and the Capability Maturity Model (CMM). Another approach for improving software quality is to stay away from the heavyweight planning-oriented processes and instead adopt agile processes and use of techniques such as RAD and eXtreme Programming (XP).

Most software engineers in the role of software architect have little or no training in the discipline of software architecture, mostly because there is no well-developed theory or standard university curriculum. As for prior generations of untrained software programmers who developed their craft through trial and error, a lot of rediscovery of principles, patterns, and techniques occurs. Practitioners and researchers began to document reusable patterns of software design and engineering processes. There is a body of knowledge accumulating in the industry and being documented as principles and patterns in books, conference proceedings, and technical journals. However,

the practicing software architect scarcely has time to keep up with the flow of information, let alone enough time to synthesize it into practical knowledge. This book is an attempt to synthesize and distill much of this information so that the practicing software architect, and especially the beginning software architect, may be able to fill in the gaps in his or her understanding of software architecture design.

The Art of Software Architecture presents software architecting independently of any particular engineering process or organization maturity. It supplies the software architect with the information and tools necessary to make sound architectural decisions and create effective software architectures. The book includes thorough introductions to and applications of methodologies; design representations and models; technologies (such as object-orientation and component-orientation); reference models; architectural frameworks; and analysis, design, and architecture patterns.

No one book can serve as a software architect's handbook. The subject is broad and deep, and it is evolving. This book focuses on how software architects create software architectures. It outlines the discipline and its methodologies and gives the reader a sense of the scope of the topic. Whereas many software architecture books focus on a process or a technology-based view, this book is organized around the fundamentals, models, and techniques of software architecture design.

This book focuses on the design methods and techniques that a software architect must practice. You cannot become a good architect by simply reading about it; you must apply the things you have learned in order to understand how they *should* be applied and how best to apply them. One barrier to effectively using object-oriented design, for example, is the skill in actually defining the right objects and their relationships. UML and object-oriented programming languages only help you express your designs; they do not help you produce good designs. Another barrier is that a solution is only as good as the problem statement. If the problem statement is confusing, wrong, or missing, then the design process has no input ("garbage in/garbage out").

The Goals of This Book

The demands of software development organizations strain software designers. This is especially true of smaller development organizations that do not have standardized development processes or a lot of experience in architecting software. These organizations make up about 70 percent of the software organizations that exist today. Most of these organizations cannot implement expansive development methodologies or adopt formal software design specification methods for any number of reasons such as cost of training in time and money, cost of tools to support the methodology, cost of evangelizing the methodology in terms of time and personal energy, the risk of introducing a

new methodology while trying to build software, and simply a lack of understanding of the practical importance of an effective software architecting process. Software development organizations need to implement practices that improve the software architecture without necessarily requiring the organization to change overnight. The software architects are often the persons who need to effect this change.

This book is especially for the software architect in the smaller, less mature software development organization (characterized as predominantly practicing *ad hoc* development). It provides practical guidance on the generation of effective software architectures. It will:

- Provide a sound understanding of the fundamental concepts of software architecture
- Serve as a road map through the information and schools of thought in software architecture
- Teach classic software architecting styles, patterns, heuristics, methodologies, and models

How This Book Is Organized

Most of the literature on software architecture addresses the structure of software but not the design processes and heuristics for generating them. Software pattern books provide a lot of help in this area because they not only show abstract software structures but they also provide some techniques for generating architectures based on these patterns. What seem to be missing are the fundamentals of software design, especially from the architecture perspective.

This book provides an integrated view of design methods, processes, practices, heuristics, and patterns and gives the reader a better sense of the scope of the topic of software architecture while providing practical guidance for designing software architectures from analysis through implementation.

In Chapter 1, "Introduction to Software Architecture," I explore the roots of software architecture. The fundamental problems of software development, which comprise the *software crisis*, are that software is expensive to develop, it is typically of low quality, and it is often delivered late. Software development has undergone several small revolutions or paradigm shifts to address these problems. Each new paradigm incorporates new technologies but still solves the problems the same basic way.

Software architecture is an emerging discipline that focuses on the design of software at a level higher than the programming language. It is possible to reason about many qualities of a software system before it is built, based on the architectural design models or *architectural description*.

In Chapter 2, "The Software Product Life Cycle," I address the role of software architecture in the software product development life cycle. There are many methodologies and views of software development, which we call development life-cycle models. Different stakeholders have different perspectives and concerns and need to see different information in order to assess progress and quality. Architecture provides another viewpoint of the life cycle that involves developing a system design that balances the competing concerns of all stakeholders.

Chapter 3, "The Architecture Design Process," presents a general model of the process of architectural design. A design solution to a problem may be a concrete artifact like source code, or it may be an abstract artifact like a high-level model. Software design is a progression of refining abstract problem statements to executable code. In the middle of this progression is a series of models that help the problem-solving process.

Design is the process of finding or discovering solutions to problems. Design methods help us search for these solutions. Models are one way to manage the complexity of design discovery. Models represent essential knowledge for solving a particular problem while suppressing other knowledge that may be irrelevant to the problem and the inclusion of which would only hinder the design process.

In Chapter 4, "Introduction to Software Design," I present the fundamental methods and techniques of software design. Software design can be viewed as a psychological activity in which a designer is applying design principles to problems in order to produce solutions. In a systematic design methodology, we reduce the risk of project failure by producing more than one possible solution; that is, we *search* for the solution. In Chapter 5, "Complexity and Modularity," precise definitions of complexity, modularity, and the notion of architectural levels of design are presented. Complexity is one of the main forces that we attempt to manage with our software development tools and methods. When not managed, complexity can cause a project to be delivered late, over budget, or cancelled. Complexity can be measured by the interconnectedness of things. In order for a system or process to exhibit complexity, it must be an aggregation of multiple interconnected parts. We refer to these connections as *dependencies.* A fundamental tool in representing a complex system, the design structure matrix (DSM), is presented.

The design structure matrix can help the architecture find the right modules for the system and the shared design decisions among modules, which are called *design rules*.

Design is about finding solutions to problems. In Chapter 6, "Models and Knowledge Representation," we see that problems and solutions are both forms of systems knowledge. In order to begin a search for a solution, we must understand the problem. There is a hierarchy of systems knowledge starting from the most basic knowledge of the types of attributes of a system, the values of those attributes, generative models that can generate those attribute

values, and finally a physical system that implements the generative model. Models are the means by which we capture and represent knowledge about the system that we are designing.

In Chapter 7, "Architecture Representation," we learn about the problems of describing the component structure of a software system. The classic views of software have fairly mature modeling notations. However, there are no standard architecture description languages that are expressive enough to represent many types of architectural styles and yet still be practical. This chapter continues the theme of models into the more concrete realm of architecture representation.

In Chapter 8, "Quality Models and Quality Attributes," I present classic system quality attributes and how the architectural design can address them. A system is understood by understanding its quality attributes. The classic software quality attribute types include functionality, security, performance, reliability, and modifiability.

In Chapter 9, "Architectural Design Principles," we learn about specific methods and techniques that can help us discover the components of the system. Design principles are applied within the context of design methods and techniques.

Chapter 10, "Applying Architectural Styles and Patterns," presents the concept of architectural style and how it influences the architecting process. Architectural styles are generalized knowledge captured about existing system architectures. There is a small set of basic architectural styles from which an architecture may be derived.

Chapter 11, "Understanding Metamodels," continues the theme of architecture models. A metamodel is a model for creating models. Well-defined metamodels can help in the discovery and creation of architectural designs by reusing domain knowledge. Reference models are metamodels that describe domain-specific problem decompositions. A reference model may be an industry standard, such as the common warehouse metamodel or the workflow reference model or an informal model presented in the software design literature. In this chapter we see how to use metamodels in the architecture process.

In Chapter 12, "Creating Architectural Descriptions," I present the IEEE Recommended Practice for the Description of Software Intensive Systems, Std. 1471. This is a standard framework for software architectural description based on the concept of multiple views.

Chapter 13, "Using Architecture Frameworks," continues with the theme of the architectural description. In this chapter I present the 4+1 View Model of Architecture and the ISO Reference Model for Open Distributed Computing (RM-ODP) as specific frameworks for creating an architectural description. The RM-ODP is a powerful model that prescribes five standard views of architecture: the enterprise viewpoint, the information viewpoint, the computational viewpoint, the engineering viewpoint, and the technology viewpoint. By following the metamodels of each of these viewpoints, the software architect can

create a series of architectural models that represent the system in various states of abstraction.

I end the book with Chapter 14, "Software Architecture Quality." In this chapter I return to the subject of quality at the architectural level of design. Quality cannot be tested into a system, so a system must be designed with quality. The candidate architecture for a system can be assessed to understand the quality attribute characteristics of the system described, before actually constructing the system. A software architecture description can be evaluated so that we may understand many potential quality attributes of the system including modifiability, performance, and reliability. Each quality attribute can be assessed using different assessment techniques.

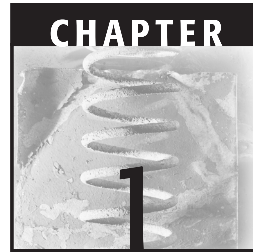
Who Should Read This Book

Beginning software architects are usually experienced software engineers. However, the software engineer must make a mental paradigm shift when it comes to designing software systems at the architectural level. All of his or her prior knowledge about object-oriented programming is still applicable, but it must be applied on a different scale, at different levels of abstraction. This book is useful for understanding how to architect a software system and even how to design a single module. The design principles can be applied at many levels of software design. Experienced software architects will find new material to broaden their knowledge and provide them with a fresh insight into software architecting.

Technical managers will gain insight into the processes of software architecting, as well as the styles of architecture and techniques used to generate them. This will enable managers to more effectively create project teams, plans, and schedules, as well as implement reuse plans, conduct design reviews, and choose an appropriate process framework. Architecture, organization, and process are interwoven. The architecture of a system influences the structure of an organization and the process by which a system is realized. Technical managers will also learn that the architecture of a system addresses many business- and development-related requirements.

Depending on what you want out of this book, you should have experience in one or more of the following:

- Object-oriented programming with a language such as C++ or Java
- Managing object-oriented projects
- Object-oriented analysis and design
- Other systems analysis and design techniques (for example, structured analysis)



Introduction to Software Architecture

Software architecture involves the integration of software development methodologies and models, which distinguishes it from particular analysis and design methodologies. The structure of complex software solutions departs from the description of the problem, adding to the complexity of software development. Software architecture is a body of methods and techniques that helps us to manage the complexities of software development.

Software architecture is a natural extension of the software engineering discipline. In early literature it was simply referred to as *programming in the large*. Software architecture presents a view of a software system as components and connectors. Components encapsulate some coherent set of functionality. Connectors realize the runtime interaction between components. The system design achieves certain qualities based on its composition from components and connectors. The architecture of a software system can be specified in a document called the architectural description. Software architecture design is not entirely different from existing software design methodologies. Rather it complements them with additional views of a system that have not been traditionally handled by methodologies like object-oriented design. We will learn that software architecture fits within a larger *enterprise architecture* that also encompasses business architecture, information technology architecture, and data architecture.

This chapter begins with a brief discussion of the evolution of software development, followed by the fundamental engineering techniques that comprise the discipline of software engineering. Finally, we look at the craft of software architecture as a discipline that complements software engineering.

Evolution of Software Development

Roughly every decade the software development field experiences a shift in software design paradigms. Design methodologies and tools must evolve as the problems and technologies become more complex. Software development was born around 1949 when the first stored-program computer, the Cambridge EDSAC, was created. Programs were initially created as binary machine instructions. This approach to programming proved to be slow and difficult because of the human inability to easily memorize long, complex binary strings. The notion of a human-readable shorthand for designing programs was conceived. Initially, the concept behind the programming shorthand was to allow a *program designer* to design a program and for a programmer or *coder* to manually translate the shorthand into binary code.

In the early 1950s, it became apparent that the majority of a programmer's time was spent correcting mistakes in software. One response to this situation was the creation of program subroutines that allowed programmers to reuse program fragments that had already been written and debugged, thus improving the productivity of programmers. By the late 1950s, the handcrafting of programs—even with the aid of reusable subroutines—was becoming uneconomical. Hence research in the area of *automatic programming* systems began. Automatic programming would allow programmers to write programs in a high-level language code, which was easier to read by humans, that would then be converted into binary machine instructions by use of another program. Thus, the first paradigm shift in software development was about to occur.

Experienced binary programmers were reluctant to change their habits to adopt a new method of working and resisted automatic programming. However, automatic programming became the dominant paradigm after International Business Machines (IBM) developed an automatic programming system for scientific programs called FORTRAN (the Formula Translator). Automatic programming not only improved programmer productivity but it also made programs portable across hardware platforms. Porting to new hardware prior to automatic programming required rewriting an entire program, which was too costly and a hindrance to selling hardware. By the mid-1960s, FORTRAN had established itself as the dominant language for scientific programming.

During the 1960s, there was a dramatic rise in the number of software development contractors and ready-made programs for specific vertical markets, such as banking and insurance. The term *software* was coined as an implicit recognition that software was viewed as an entity in its own right. Software was also being marketed and sold separately from hardware, which marked a departure from the earlier practices of giving software away for free as part of the hardware platform. The hiding of the internal details of an operating system using abstract programming interfaces improved programmer productivity and helped make programs more portable across hardware platforms. Programs could work with logical files instead of physical locations of bits on a tape or magnetic disk. It was also during this period that extensive research began in programming languages, which continued through the 1970s.

By the late 1960s, it was clear that software development was unlike the construction of physical structures: You couldn't simply hire more programmers to speed up a lagging development project (Brooks, 1975). Software had become a critical component of many systems, yet was too complex to develop with any certainty of schedule or quality. This imposed financial and public safety concerns. The situation became known as the *software crisis*, and in response the software development community instituted *software engineering* as a discipline. It called for software manufacturing to be based on the same types of theoretical foundations and practical disciplines that are traditional for the established branches of engineering.

In 1968, Edsger Dijkstra published a paper on the design of a multiprogramming system called "THE" (Dijkstra, 1968). This is one of the first papers to document the design of a software system using hierarchical layers, from which the phrase *layers of abstraction* was derived. Dijkstra organized the design of the system in layers in order to reduce the overall complexity of the software. Though the term *architecture* had not yet been used to describe software design, this was certainly the first glimpse of software architecture; programming in the large was a common phrase used to describe this aspect of software design.

A second paradigm shift occurred in the first half of the 1970s with the development of structured design and software development models. These were based on a more organic, evolutionary approach, departing from the waterfall-based methodologies of hardware engineering. Research into quantitative techniques for software design began but never established itself in mainstream industry, in part due to the inherent qualitative nature of software systems. During this time researchers began focusing on software design to address the problems of developing complex software systems. The premise of this work was that software design is a separate activity from implementation in software development and that it requires its own tools, techniques, and modeling languages.

In 1972 David Parnas published a paper that discussed how modularity in systems design could improve system flexibility and comprehensibility while shortening development time (Parnas, 1972). He introduced the programming world to the concept of *information hiding*, which is one of the most fundamental design principles in software development today.

In the 1980s, software engineering research shifted focus toward integrating designs and design processes into the larger context of software development process and management. Structured design methods could not scale as software systems grew in complexity, and in the latter half of the 1980s a new design paradigm began to take hold—*object-orientation*. With object-oriented programming, software engineers could (in theory) model the problem domain and solution domain within an implementation language. Research that led to object orientation can be traced back to the late 1960s with the development of Simula, a simulation programming language, and it was later refined in Smalltalk. Object-oriented programming started to become popular with C++. At this time there was also a shift in application design metaphors from text-based terminals to graphical user interfaces (GUIs). Object-oriented programming was well suited for the development of GUIs. In the late 1980s and early 1990s, the term *software architecture* began to appear in literature.

Object-oriented programming was in full swing by the mid-1990s, when the Internet became the new computing platform. At around the same time, software design was experiencing another shift. This time it was not away from the prior design paradigms, however, but rather toward an integration of methods. Object orientation was being augmented with design techniques such as Class/Responsibilities/Collaborators (CRC) cards and use case analysis. Methods and modeling notations that came out of the structured design movement were making their way into the object-oriented modeling methods. This included diagramming techniques such as state transition diagrams and processing models.

It was becoming obvious that an integrated, multiviewed approach to design was required to manage the complexity of designing and developing large-scale software systems. This multiview approach culminated in the development of the Unified Modeling Language (UML), which integrates modeling concepts and notations from many methodologists. It was also during the late 1990s that design patterns started becoming a popular way to share design knowledge.

I believe that we are experiencing a fifth paradigm shift in software development, which is the recognition that software architecture is an important aspect of software development and of the introduction of software architecture methods and activities into the software development life cycle. This shift, like the last one, is not one of divergence of design methods but rather one of the integration of new methods and activities with existing methods and activities.