

SOFTWARE ARCHITECTURE

FOUNDATIONS, THEORY, AND PRACTICE

Richard N. Taylor
Nenad Medvidović
Eric M. Dashofy

WILEY
STUDENT
EDITION

RESTRICTED!
FOR SALE ONLY IN
INDIA, BANGLADESH, NEPAL,
PAKISTAN, SRI LANKA
& BHUTAN

WILEY

SOFTWARE ARCHITECTURE

Foundations, Theory, and Practice

Richard N. Taylor
University of California, Irvine

Nenad Medvidović
University of Southern California

Eric M. Dashofy
The Aerospace Corporation

WILEY

To Lily May, Lyrica, and Richard
- *Richard N. Taylor*

To Pavle and Sasha
- *Nenad Medvidović*

To Mom and Keith
- *Eric M. Dashofy*

Software Architecture: Foundations, Theory, and Practice

Authorized reprint by Wiley India Pvt. Ltd., 4435-36/7, Ansari Road, Daryaganj,
New Delhi - 110002.

Copyright © 2005, 2006 by John Wiley & Sons, Inc. All rights reserved.

All rights reserved. AUTHORIZED REPRINT OF THE EDITION PUBLISHED
BY JOHN WILEY & SONS, U.K. No part of this book may be reproduced in any
form without the written permission of the publisher.

Limits of Liability/ Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or website is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that Internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books. For more information about Wiley products, visit our website at: www.wiley.com.

Reprint: 2014

Printed at: Printed in India by Artxel,

ISBN: 978-81-265-2802-8

SOFTWARE ARCHITECTURE

Foundations, Theory, and Practice

Richard N. Taylor
University of California, Irvine

Nenad Medvidović
University of Southern California

Eric M. Dashofy
The Aerospace Corporation

WILEY

CREDITS

Figure 10-2 from Figure 1 of Software Architectural Support for Handheld Computing. Medvidović, N.; Mikic-Rakic, M.; Mehta, N.R.; Malek, S. IEEE Computer, 36 (9):66–73, 2003. © IEEE. Used by permission.

Figure 10-9 used by permission of Antonio Carzaniga.

Ch. 11 Sidebar: “Skype on Skype” © 2008 Skype Limited. Used by permission.

Figure 13-1 from Table 1 in A Call to Action: Look Beyond the Horizon. J. M Wing, IEEE Security & Privacy, 1 (6): 62–67, 2003. © IEEE Used by permission.

Figure 14-1 Diagram by Donald Ryan, from HOW BUILDINGS LEARN by Stewart Brand, copyright © 1994 by Stewart Brand. Used by permission of Viking Penguin, a division of Penguin Group (USA) Inc.

Figure 14-5 from Figure 2 of An Architecture-Based Approach to Self-Adaptive Software. Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Greg Johnson, Nenad Medvidović, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. IEEE Intelligent Systems 14 (3): 54–62. 1999 © IEEE. Used by permission.

Ch. 14 Sidebar: Autonomic Computing. Reprint Courtesy of International Business Machines Corporation, copyright 2001 © International Business Machines Corporation.

Ch. 14 Sidebar: Quiescence. Text excerpt from Kramer, J. and Magee, J., “Change Management of Distributed Systems,” Proceedings of the 3rd ACM SIGOPS European Workshop: Autonomy or interdependence in distributed systems? (1988) <http://doi.acm.org/10.1145/504092.504113>: Copyright © 1988 ACM, Inc. Reprinted by permission.

Portion of Figure 15-14 from Apollo Operations Handbook, Lunar Module LM5 and Subsequent, Volume 1 Subsystems Data (Document LMA790-3-LM5), 1968.

Figures 16-6 & 16-7 Copyright © The Open Group. Used with permission. TOGAF™ is a Trademark of The Open Group.

Figure 16-14 Kroll/Krutchen, RATIONAL UNIFIED PROCESS MADE EASY: PRACT, (figure 1.2) © Pearson Education, Inc. Reproduced by permission of Pearson Education, Inc.

Contents

1	The Big Idea	1
1.1	The Power of Analogy: The Architecture of Buildings, 2	
1.1.1	Limitations of the Analogy, 5	
1.1.2	So, What's the Big Idea?, 6	
1.2	The Power and Necessity of Big Ideas: The Architecture of the Web, 7	
1.3	The Power of Architecture in the Small: Architecture on the Desktop, 12	
1.4	The Power of Architecture in Business: Productivity and Product Lines, 15	
1.5	End Matter, 20	
1.6	Review Questions, 21	
1.7	Exercises, 21	
1.8	Further Reading, 21	
2	Architectures in Context: The Reorientation of Software Engineering	23
2.1	Fundamental Understandings, 24	
2.2	Requirements, 25	
2.3	Design, 29	
2.3.1	Design Techniques, 31	
2.4	Implementation, 33	
2.4.1	Implementation Strategies, 35	
2.5	Analysis and Testing, 38	
2.6	Evolution and Maintenance, 40	
2.7	Processes, 42	
2.7.1	The Turbine Visualization, 43	
2.7.2	Example Process Depictions, 49	
2.8	End Matter, 52	
2.9	Review Questions, 53	
2.10	Exercises, 54	
2.11	Further Reading, 55	
3	Basic Concepts	57
3.1	Terminology, 58	
3.1.1	Architecture, 58	
3.1.2	Component, 68	
3.1.3	Connector, 70	
3.1.4	Configuration, 72	
3.1.5	Architectural Style, 72	
3.1.6	Architectural Pattern, 73	

- 3.2 Models, 75
- 3.3 Processes, 75
- 3.4 Stakeholders, 79
- 3.5 End Matter, 79
- 3.6 Review Questions, 80
- 3.7 Exercises, 80
- 3.8 Further Reading, 81

4 Designing Architectures

83

- 4.1 The Design Process, 85
- 4.2 Architectural Conception, 87
 - 4.2.1 Fundamental Conceptual Tools, 87
 - 4.2.2 The Grand Tool: Refined Experience, 90
- 4.3 Refined Experience in Action: Styles and Architectural Patterns, 91
 - 4.3.1 Domain Specific Software Architectures, 93
 - 4.3.2 Architectural Patterns, 94
 - 4.3.3 Introduction to Styles, 99
 - 4.3.4 Simple Styles, 102
 - 4.3.5 More Complex Styles, 124
 - 4.3.6 Discussion: Patterns and Styles, 137
 - 4.3.7 Design Recovery, 139
- 4.4 Architectural Conception in Absence of Experience: Unprecedented Design, 144
- 4.5 Putting it All Together: Design Processes Revisited, 149
 - 4.5.1 Insights from Requirements, 150
 - 4.5.2 Insights from Implementation, 151
- 4.6 End Matter, 152
- 4.7 Review Questions, 153
- 4.8 Exercises, 154
- 4.9 Further Reading, 154

5 Connectors

157

- 5.1 Connectors in Action: A Motivating Example, 159
- 5.2 Connector Foundations, 161
- 5.3 Connector Roles, 163
- 5.4 Connector Types and Their Variation Dimensions, 164
 - 5.4.1 Procedure Call Connectors, 165
 - 5.4.2 Event Connectors, 166
 - 5.4.3 Data Access Connectors, 167
 - 5.4.4 Linkage Connectors, 168
 - 5.4.5 Stream Connectors, 169
 - 5.4.6 Arbitrator Connectors, 170
 - 5.4.7 Adaptor Connectors, 170
 - 5.4.8 Distributor Connectors, 171

5.5	Example Connectors, 172	
5.5.1	Event-Based Data Distribution Connectors, 173	
5.5.2	Grid-Based Data Distribution Connectors, 174	
5.5.3	Client-Server-Based Data Distribution Connectors, 175	
5.5.4	P2P-Based Data Distribution Connectors, 176	
5.6	Using the Connector Framework, 176	
5.6.1	Selecting Appropriate Connectors, 177	
5.6.2	Detecting Mismatches, 180	
5.7	End Matter, 181	
5.8	Review Questions, 182	
5.9	Exercises, 183	
5.10	Further Reading, 183	
6	Modeling	185
6.1	Modeling Concepts, 186	
6.1.1	Stakeholder-Driven Modeling, 186	
6.1.2	Basic Architectural Concepts, 188	
6.1.3	Elements of the Architectural Style, 189	
6.1.4	Static and Dynamic Aspects, 190	
6.1.5	Functional and Non-Functional Aspects, 191	
6.2	Ambiguity, Accuracy, and Precision, 191	
6.2.1	Ambiguity, 192	
6.2.2	Accuracy and Precision, 192	
6.3	Complex Modeling: Mixed Content and Multiple Views, 194	
6.3.1	Views and Viewpoints, 194	
6.3.2	Consistency among Views, 196	
6.4	Evaluating Modeling Techniques, 198	
6.5	Specific Modeling Techniques, 199	
6.5.1	Generic Techniques, 199	
6.5.2	Early Architecture Description Languages, 209	
6.5.3	Domain- and Style-Specific ADLs, 221	
6.5.4	Extensible ADLs, 229	
6.6	When Systems Become Too Complex to Model, 242	
6.7	End Matter, 243	
6.8	Review Questions, 246	
6.9	Exercises, 246	
6.10	Further Reading, 246	
7	Visualization	249
7.1	Visualization Concepts, 250	
7.1.1	Canonical Visualizations, 250	
7.1.2	Textual Visualizations, 251	
7.1.3	Graphical Visualizations, 253	
7.1.4	Hybrid Visualizations, 255	
7.1.5	The Relationship between Visualizations and Views, 255	

- 7.2 Evaluating Visualizations, 259
 - 7.2.2 Constructing a Visualization, 261
 - 7.2.3 Coordinating Visualizations, 264
 - 7.2.4 Beyond Design: Using Visualization Dynamically, 266
- 7.3 Common Issues in Visualization, 268
 - 7.3.1 Same Symbol, Different Meaning, 268
 - 7.3.2 Differences without Meaning, 269
 - 7.3.3 Decorations without Meaning, 269
 - 7.3.4 Borrowed Symbol, Different Meaning, 270
- 7.4 Evaluating Visualization Techniques, 272
- 7.5 Techniques, 272
 - 7.5.1 Textual Visualizations, 272
 - 7.5.2 Informal Graphical Editors, 274
 - 7.5.3 UML: The Unified Modeling Language, 276
 - 7.5.4 Rapide, 280
 - 7.5.5 The Labeled Transition State Analyzer (LTSA), 282
 - 7.5.6 xADL 2.0, 284
- 7.6 End Matter, 288
- 7.7 Review Questions, 289
- 7.8 Exercises, 290
- 7.9 Further Reading, 290

8 Analysis

291

- 8.1 Analysis Goals, 295
 - 8.1.1 Completeness, 295
 - 8.1.2 Consistency, 296
 - 8.1.3 Compatibility, 302
 - 8.1.4 Correctness, 303
- 8.2 Scope of Analysis, 303
 - 8.2.1 Component- and Connector-Level Analysis, 304
 - 8.2.2 Subsystem- and System-Level Analysis, 305
 - 8.2.3 Data Exchanged in the System or Subsystem, 307
 - 8.2.4 Architectures at Different Abstraction Levels, 308
 - 8.2.5 Comparison of Two or More Architectures, 310
- 8.3 Architectural Concern being Analyzed, 310
- 8.4 Level of Formality of Architectural Models, 312
- 8.5 Type of Analysis, 313
- 8.6 Level of Automation, 314
- 8.7 System Stakeholders, 315
- 8.8 Analysis Techniques, 317
 - 8.8.1 Inspections and Reviews, 317
 - 8.8.2 Model-Based Analysis, 322
 - 8.8.3 Simulation-Based Analysis, 328
- 8.9 End Matter, 333
- 8.10 Review Questions, 334

- 8.11 Exercises, 335
- 8.12 Further Reading, 336

9 Implementation 337

- 9.1 Concepts, 338
 - 9.1.1 The Mapping Problem, 338
 - 9.1.2 Architecture Implementation Frameworks, 340
 - 9.1.3 Evaluating Frameworks, 343
 - 9.1.4 Middleware, Component Models, and Application Frameworks, 343
 - 9.1.5 Building a New Framework, 346
 - 9.1.6 Concurrency, 347
 - 9.1.7 Generative Technologies, 348
 - 9.1.8 Ensuring Architecture-to-Implementation Consistency, 349
- 9.2 Existing Frameworks, 350
 - 9.2.1 Frameworks for the Pipe-and-Filter Architectural Style, 350
 - 9.2.2 Frameworks for the C2 Architectural Style, 352
- 9.3 Examples, 360
 - 9.3.1 Implementing Lunar Lander in the Pipe-and-Filter Style Using the java.io Framework, 360
 - 9.3.2 Implementing Lunar Lander in the C2-Style Using the Lightweight C2 Framework, 366
- 9.4 End Matter, 377
- 9.5 Review Questions, 378
- 9.6 Exercises, 378
- 9.7 Further Reading, 379

10 Deployment and Mobility 381

- 10.1 Overview of Deployment and Mobility Challenges, 385
- 10.2 Software Architecture and Deployment, 387
 - 10.2.1 Basic Concepts, 388
 - 10.2.2 Deployment Activities, 388
 - 10.2.3 Tool Support, 402
- 10.3 Software Architecture and Mobility, 405
 - 10.3.1 Basic Concepts, 405
 - 10.3.2 Mobility Paradigms, 405
 - 10.3.3 Challenges in Migrating Code, 406
- 10.4 End Matter, 408
- 10.5 Review Questions, 409
- 10.6 Exercises, 409
- 10.7 Further Reading, 410

11 Applied Architectures and Styles 413

- 11.1 Distributed and Networked Architectures, 414
 - 11.1.1 Limitations of the Distributed Systems Viewpoint, 415
- 11.2 Architectures for Network-Based Applications, 416

11.2.1	The REpresentational State Transfer Style (REST), 416	
11.2.2	Commercial Internet-Scale Applications, 422	
11.3	Decentralized Architectures, 424	
11.3.1	Shared Resource Computation: The Grid World, 425	
11.3.2	Peer-to-Peer Styles, 426	
11.3.3	Summary Notes on Latency and Agency, 432	
11.4	Service-Oriented Architectures and Web Services, 433	
11.5	Architectures from Specific Domains, 437	
11.5.1	Robotics, 437	
11.5.2	Wireless Sensor Networks, 443	
11.6	End Matter, 443	
11.7	Review Questions, 445	
11.8	Exercises, 445	
11.9	Further Reading, 446	
12	Designing for Non-Functional Properties	447
12.1	Efficiency, 450	
12.1.1	Software Components and Efficiency, 451	
12.1.2	Software Connectors and Efficiency, 453	
12.1.3	Architectural Configurations and Efficiency, 455	
12.2	Complexity, 459	
12.2.1	Software Components and Complexity, 460	
12.2.2	Software Connectors and Complexity, 463	
12.2.3	Architectural Configurations and Complexity, 464	
12.3	Scalability and Heterogeneity, 467	
12.3.1	Software Components and Scalability, 468	
12.3.2	Software Connectors and Scalability, 471	
12.3.3	Architectural Configurations and Scalability, 473	
12.4	Adaptability, 475	
12.4.1	Software Components and Adaptability, 475	
12.4.2	Software Connectors and Adaptability, 476	
12.4.3	Architectural Configurations and Adaptability, 477	
12.5	Dependability, 478	
12.5.1	Software Components and Dependability, 480	
12.5.2	Software Connectors and Dependability, 480	
12.5.3	Architectural Configurations and Dependability, 481	
12.6	End Matter, 483	
12.7	Review Questions, 484	
12.8	Exercises, 484	
12.9	Further Reading, 485	
13	Security and Trust	487
13.1	Security, 489	
13.2	Design Principles, 492	
13.3	Architectural Access Control, 497	

13.3.1	Access Control Models, 498	
13.3.2	Connector-Centric Architectural Access Control, 499	
13.4	Trust Management, 508	
13.4.1	Trust, 509	
13.4.2	Trust Model, 510	
13.4.3	Reputation-Based Systems, 510	
13.4.4	Architectural Approach to Decentralized Trust Management, 514	
13.5	End Matter, 522	
13.6	Review Questions, 523	
13.7	Exercises, 523	
13.8	Further Reading, 524	
14	Architectural Adaptation	525
14.1	Concepts of Architecture-Centric Adaptation, 526	
14.1.1	Sources and Motivations for Change, 526	
14.1.2	Shearing Layers, 529	
14.1.3	Structural Elements Subject to Change, 532	
14.1.4	Change Agents and Context, 535	
14.1.5	Architecture: The Central Abstraction, 539	
14.2	A Conceptual Framework for Architectural Adaptation, 539	
14.3	Techniques for Supporting Architecture-Centric Change, 542	
14.3.1	Basic Techniques Corresponding to Activities of the Conceptual Framework, 542	
14.3.2	Architectures/Styles that Support Adaptation, 551	
14.3.3	The Special Problems of On-the-Fly and Autonomous Adaptation, 557	
14.4	End Matter, 561	
14.5	Review Questions, 562	
14.6	Exercises, 563	
14.7	Further Reading, 563	
15	Domain-Specific Software Engineering	565
15.1	Domain-Specific Software Engineering in a Nutshell, 570	
15.1.1	Similar Problems, Similar Solutions, 570	
15.1.2	Viewing DSSE through the Prism of Domain, Business, and Technology, 573	
15.2	Domain-Specific Software Architecture, 574	
15.2.1	Domain Knowledge, 574	
15.2.2	Canonical Requirements, 585	
15.2.3	Canonical Solution Strategies—Reference Architectures, 588	
15.2.4	Product Lines and Architecture, 591	
15.2.5	Product-Line Concepts, 592	
15.2.6	Specifying the Architecture of a Product Line, 594	
15.2.7	Capturing Variations Over Time, 559	
15.2.8	Using Product Lines as Tools for What-If Analysis, 600	
15.2.9	Implementing Product Lines, 602	

- 15.2.10 Unifying Product Architectures with Different Intellectual Heritage, 604
- 15.2.11 Organizational Issues in Creating and Managing Product Lines, 606
- 15.3 DSSAs, Product Lines, and Architectural Styles, 606
- 15.4 DSSE Examples, 608
 - 15.4.1 Koala and Consumer Electronics, 608
 - 15.4.2 Software-Defined Radios, 611
- 15.5 End Matter, 616
- 15.6 Review Questions, 618
- 15.7 Exercises, 618
- 15.8 Further Reading, 619

16 Standards

621

- 16.1 What Are Standards?, 622
 - 16.1.1 Why Use Standards?, 623
 - 16.1.2 Drawbacks of Standards, 624
 - 16.1.3 When to Adopt, 625
- 16.2 Specific Standards, 627
 - 16.2.1 Conceptual Standards, 627
 - 16.2.2 Notational Standards, 639
 - 16.2.3 SysML, 641
 - 16.2.4 Standard Tools, 645
 - 16.2.5 Telelogic System Architect, 647
- 16.3 Process Standards, 648
 - 16.3.1 Rational Unified Process, 648
 - 16.3.2 Model-Driven Architecture, 650
- 16.4 End Matter, 651
- 16.5 Review Questions, 653
- 16.6 Exercises, 653
- 16.7 Further Reading, 654

17 People, Roles, and Teams

655

- 17.1 Who Are Software Architects?, 657
 - 17.1.1 Architect as a Software Designer, 658
 - 17.1.2 Architect as a Domain Expert, 658
 - 17.1.3 Architect as a Software Technologist, 659
 - 17.1.4 Architect as a Standards Compliance Expert, 660
 - 17.1.5 Architect as a Software Engineering Economist, 661
 - 17.1.6 Some Bad Habits, 661
- 17.2 What Do Software Architects Do?, 663
 - 17.2.1 Develop Project Strategy, 663
 - 17.2.2 Design Systems, 664
 - 17.2.3 Communicate with Stakeholders, 664
 - 17.2.4 Lead, 664
- 17.3 How Do Software Architects Work?, 665

17.3.1 Balance of Skills, 665	
17.3.2 Allegiance to the Project, 666	
17.3.3 Allegiance to the Organization, 667	
17.3.4 Duration of Involvement, 668	
17.3.5 Team Structure, 669	
17.4 How Do Software Architects Relate to Other Stakeholders?, 669	
17.4.1 Architects and Engineers, 670	
17.4.2 Architects and Managers, 671	
17.4.3 Other Stakeholders, 671	
17.5 Remaining Challenges, 673	
17.6 End Matter, 673	
17.7 Review Questions, 674	
17.8 Further Reading, 674	
Bibliography	675
Index	691

About the Cover

THE ARCHITECT'S DREAM, THOMAS COLE, 1840

In *The Architect's Dream*, Thomas Cole painted a vision of the history of Western civilization as a succession of buildings through time and space. A massive Egyptian pyramid and temple, Roman aqueduct, and Greek and Roman temples demonstrate his knowledge of architectural history, likely taken from architectural pattern books given to the artist as partial payment for the commission, showing the range of forms that inspired 19th-century architects. Geographical and historical contrasts separate the right and left sides of the composition. Tall trees mirror the cathedral's steeple, pointing towards Heaven, indicating the inspiration of nature in the architectural form of the church. A sacred place in harmony with nature and God, the shaded church contrasts with the towering classical structures that appear artificial and warn against fleeting power. Inspired by Salisbury Cathedral and Chapter House in England, the cathedral suggests the 19th-century Gothic architectural revival in America. Symmetrical capitals frame the sides of the canvas, their forms taken from Salisbury Chapter House, creating a Gothic window. We gaze through the theatrical window, to behind the cathedral spire, where the setting sun shines brilliant light through the windows, creating the impression of stained glass, and evoking divine light. Cole's youth and travels in England may have contributed to a sense of national pride that influenced his reference to an English Gothic cathedral, an architectural style from shared American and English history, thereby extending American history back in time. With his face turned towards the radiant light emanating through the cathedral steeple, the architect reclines on gigantic pattern books, the catalyst of this vision of creative imagination. By including architectural tools, books, and plans alongside the dreaming architect, the artist indicates the importance of engineering and design in growing American commerce, elevating the artist and architect's roles. The extraordinary variety and enduring beauty of Thomas Cole's painting celebrates human creativity.

The painting provides an apt illustration for a textbook on software architecture. Many contemporary software systems have designs that draw heavily from the designs of prior applications. Others take fresh inspiration from social systems or nature. Just as Cole's architect is surrounded by past structures, books, and plans, so the skilled software architect has existing systems and, importantly, a vast store of styles, techniques, processes, and tools from which to choose when confronting a design challenge.

—LET & RNT

Credits

Figures 1-4, 1-5, 1-6, and 1-7 courtesy Royal Philips N.V.

Figure 2-13 adapted from Figure 1 from: Weaving Together Requirements and Architectures, Bashar Nuseibeh. IEEE Computer 34 (2):115–117. 2001. © IEEE. Used by permission.

Ch. 4 Sidebar: “Styling with Perry and Wolf” Text excerpt from Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17 (4): 40–52. 1992. © Dewayne E. Perry and Alexander L. Wolf. Used by permission.

Figure 6-1 adapted from Maier and Rechtin, The Art of Software Architecting, 2nd. Ed. Copyright © CRC Press. Used with permission.

Figure 7-18 used by permission of Jeff Magee.

Figures 8-4, 12-4, and 12-5 adapted from Figures 1, 5: Bowman, R.T. et al, “Linux as a Case Study: Its Extracted Software Architecture,” Proceedings of the 21st ACM/IEEE Conference on Software Engineering (ICSE’99), pp. 555–563. DOI: <http://doi.acm.org/10.1145/302405.302691>; Copyright © 1999 ACM, Inc. Reprinted by permission.

The ATAM Figure located at page 320 in this publication has been adapted from the ATAM diagram © 2008 by Carnegie Mellon University and is reproduced in its adapted form in this publication with special permission from the Software Engineering Institute of Carnegie Mellon University. ® Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ANY CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK OR COPYRIGHT INFRINGEMENT.

Neither Carnegie Mellon University nor the Software Engineering Institute directly or indirectly endorse this publication. Accuracy and interpretation of figure 8-12 are the responsibility of Eric Dashofy. The Software Engineering Institute has not participated in the creation of Figure 8-12.

Figure 10-1 adapted from Figure 1; Figure 11-12 adapted from Figure 4; both from Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. Sam Malek, Chiyoung Seo, Sharmila Ravula, Brad Petrus, and Nenad Medvidović. Proceedings of the Twenty-Ninth International Conference on Software Engineering, IEEE Computer Society, pp. 591–601. 2007. © IEEE. Used by permission.

Preface

Software architecture is the centerpiece of modern system development. The goal of architecture-centric development is the effective, efficient, competitive development of software products. The activities of development are anchored in the architecture. The goal of this text is to provide both student and professional a comprehensive treatment of architecture-centric development, instructing how to develop products and serving as a reference for the panoply of techniques, modeling notations, standards, and methods comprising the approach.

The text is intended for upper-division undergraduate and graduate courses in software architecture, software design, component-based software engineering, and distributed systems; the text may also be used in introductory as well as advanced software engineering courses. The targeted courses can be of different lengths, from a single ten-week quarter, to a fifteen-week semester, to a two-semester sequence. Both the breadth and depth of topics covered will vary depending on the exact nature of the course. Specific recommendations for instructors are provided below.

With such an ambitious goal, the text is extensive and may be approached in different ways. In this preface we guide the reader in the use of the book. Hence we briefly present here:

- The general view of software architecture taken by the book.
- The scope of the book.
- The intended audiences.
- Our assumptions regarding the technical maturity/background of the professionals and students who will use the book.
- Thoughts regarding selection of materials for use in the classroom, and by professionals, including systems engineers, software architects, and managers of software systems development.
- How instruction in the material may be supported by tools.
- A summary of each chapter.

SOFTWARE ARCHITECTURE

The text adopts a particular definition of software architecture: the set of principal design decisions governing a system. This definition is broad and stakeholder-centric. Nonetheless, a decidedly technical view of software architecture pervades the book. We believe that software architecture must help developers create real implemented systems. Software architecture without a tie to implementation, deployment, and long-term adaptation misses its potential by a wide mark and risks being irrelevant busywork; tolerated perhaps, but not treasured.

The comprehensive character of software architecture as presented in the text makes it, in our estimation, the central focus of software engineering. The text attempts to show

architecture's critical role in software engineering, yet the text does not attempt to be a general software engineering reference. There are, for example, many important parts of software engineering not treated at all, such as source-code testing strategies, general project management, and development environments.

THE SCOPE OF THE BOOK

This book is characterized by breadth. Rather than focus on one method, notation, tool, or process, the book attempts to widely survey software development from an architectural perspective, putting many of these elements in context, comparing and contrasting them with one another. A reader should expect to gain a broad appreciation for the role of software architecture across the wide variety of activities that characterize the development of large, software-intensive systems.

Our assumption is that readers seek fundamental insights and knowledge to supplement their own experience, rather than canned solutions to specific problems (those, if they do exist, can most likely be found elsewhere). The text therefore focuses on overarching principles, general strategies, and broadly applicable techniques. It illustrates them with many specific examples from various domains. The text does not advocate a single approach and it is not a manual or advertisement for one—even the authors' own.

Audiences

The book attempts to address the needs of several audiences.

The *instructor* audience is supported by a detailed treatment of a broad spectrum of software architecture-related topics, from which different subsets can be selected for use in courses with specific foci. An instructor will be able to tailor the material provided in the book to both the length and the objective of a course. To date, the authors have used this text in five quarter-long (ten-week) courses and two semester-long (fifteen-week) courses, at both the undergraduate and graduate levels. The courses have focused on software engineering, software architecture, software design, software interoperability, and distributed systems. Both semester-long courses have used the entire text, while the quarter-long courses have relied on different selections of the chapters. The instructor using the text will be able to rely on a comprehensive set of lecture slides, solutions to end-of-chapter exercises, sample exams with solutions, and a tool suite that will be described below.

The *student* audience is supported with a presentation that is comprehensive, making relatively few assumptions about background knowledge and experience. We do assume that the student is generally familiar with the most basic elements of software engineering and programming. The order of the text and the way the material is presented makes the student the primary audience of the book—the other audiences will pick and choose material from various parts of the text.

The *professional* audience is supported with, for example, comprehensive treatment of the major modeling techniques, architectural description standards, and the way they are directly tied to the implementation of software products. Our perspective is not confined to some presumed ivory tower: The standards and notations include commercial, government, and research products. The text also attempts to provide, in the majority of chapters, specific guidance on how to achieve various goals, such as how to design to achieve specific system properties. References and examples draw from a wide range of industrial experience.

Professionals should expect to encounter realistic and frank assessments of the techniques, notations, and tools with which they are already familiar, while also being exposed to projects and approaches that provide valuable insights but are nonetheless not well-known in practitioners' circles.

The *systems engineer* will find that this book's perspective, while still focused on software, has implications that are quite a bit broader. Insights in this book will enable better understanding of the unique issues involved in the development of large *software* systems. Many of the insights, however, are applicable to *systems* development in general. Insights, notations, and methods from the systems engineering and systems architecture community are not ignored, and this book explores overlap between systems and software engineering throughout.

The systems and/or software development *manager* will find substantial material here that addresses the way in which architecture-centric development can shape and improve processes and organizations, especially with regard to the development of product families.

ASSUMPTIONS AND BACKGROUND

Depending on the sections of the book used for reference or in class, the necessary background varies. As a whole, however, the book assumes familiarity with the basics of software development. This includes the core ideas of software development activities and processes, notions of requirements, design, programming, and analysis and testing. The level of maturity required is, in some sense, only what would be acquired in a single, quarter- or semester-long introduction to software development/engineering. Appreciation for the problems of large-scale development is very useful. Proficiency in any particular programming language, development environment, or modeling tool such as UML is not required.

Additional experience will likely make it possible for certain readers to develop a deeper understanding and appreciation of specific topics. For example, the implementation chapter will be of most value to those with substantive training and experience in programming. Similarly, appreciating the discussion of deployment or product lines demands some level of understanding of how commercial development and product-and-revenue focused businesses work. However, none of the chapters presuppose an extensive background in the area on which they focus.

HOW THE BOOK CAN BE USED

The text is designed to be used as a textbook in formal classroom settings, both at the undergraduate and graduate levels. It can also be used as a reference for systems engineers, software architects, managers, and programmers. Chapters 1 through 5 present the critical concepts of software architecture. Chapter 5 also bridges to the next section, the nuts and bolts of applying the concepts, which encompasses Chapters 6 through 10. Chapters 11 through 17 treat associated issues: topics that may be critical in one development context, but not in another.

Classroom Usage

There is adequate material in the book for a year-long class at the master's degree level. The presentation does not presuppose that this is the normative case however. The authors

have used material from the book in quarter-long and semester-long courses, including, as a primary text, an undergraduate software engineering course and a graduate software engineering course. The book, in its preprint form, has also been used at several other universities, including Drexel University, Southern Methodist University, Seattle University, and the University of Zurich.

Clearly, material has to be chosen based upon the level of the instruction and the length of the class. Nonetheless, a few comments are in order. The concepts section—Chapters 1 through 5—is essential to use of the book. The material in the subsequent chapters depends on these core concepts. How long it will take to cover those initial chapters, however, will vary significantly. The first chapter, for instance, will be a very quick read for the experienced. Similarly, Chapter 2 will be relatively quick for those with experience in software engineering, but will take a bit longer in an undergraduate class that has minimal prerequisites. Chapter 4 is a long chapter, but it covers a single, critical topic: how to go about designing a system. This chapter may thus be a major focus over many class sessions, especially for those with little training or experience in design.

Selection of materials from Chapters 6 through 10 will depend upon the level of the class, the course objectives, and the length of the class. The authors recommend a shallow treatment of all of them over a deep look at any individual chapters, but this is up to the instructor.

Several suggested pathways through the book, depending on course type and length, are as follows:

- A fifteen-week software architecture course—Chapters 1 through 17
- A ten-week software architecture course—Chapters 1 through 9, 11, 12, and 15
- A ten-week distributed systems course—Chapters 1 through 6, 7 through 15, and 17
- A fifteen-week software engineering course—Chapters 1 through 15 and 17
- A ten-week software engineering course—Chapters 1 through 6, 8, 9, 11, and 12

If the book is used as a primary text for a software engineering course, the instructor may want to supplement it with material covering topics such as software testing, OO design patterns, software engineering economics, and project management. The book can also be used as a companion text in a software analysis and quality assurance course; the relevant chapters would be Chapters 8, 12, 13 and 14.

A key issue the instructor will have to deal with is use of the exercises. Each chapter has a number of exercises listed. Software architecture is not a topic like algebra where numerous simple questions can be posed and the answers given in the back of the book. Software architecture concerns itself with design, with complex systems, and with long-lived products. Hence it is difficult to pose exercises that fit nicely into the constraints of a modern college class. Many of the exercises included in the text could be assigned as mini-projects. The instructor will thus have to tailor the scope of the assignments and provide guidance to the students regarding the level of detail expected in the answers. Assessing the answers similarly will require thought and discretion.

Use by Professionals

The authors have tried to include substantial material to make the book a superior reference for the professional. A theme running through the entire book is software reuse and, more generally, reuse of corporate and domain knowledge. Similarly, there is explicit

reliance in the presented techniques on the use of frameworks to help achieve rapid system implementation. A wide range of modeling notations is discussed, and a complete chapter on standards is included to provide a thoughtful basis for choosing a notation or approach for a project. Similarly, the discussion of designing for non-functional properties is targeted at providing proven techniques capable of addressing these difficult issues. Chapter 4 on designing similarly will be useful as a reference to the professional. Without being a handbook of field techniques, it nevertheless presents a wide range of architectural styles and design techniques, with the styles summarized individually and in a comparison table to aid the professional in choosing the right technique for the job at hand.

Tool Support

Use of the book in the classroom or by professionals in practice does not require the use of any particular tool or programming language. Clearly, however, the subject is designing and building applications—products—and that only happens effectively in the presence of good tools. The professional may be constrained by the choice of the modeling notation required in a particular development context. If AADL, for example, is mandated as the architecture description language for a project, then the die is cast.

For those contexts where external constraints do not dictate, the instructor or reader is encouraged to use technologies that not only help model an architecture, but build implementations based upon those models. If modeling alone is supported, then the value of a tool over a simple boxes-and-arrows drawing tool rests upon the value obtained from analyses that can be performed on the model. If building real products is the goal, then a tool suite that addresses the full range of developers' needs is required.

To this latter end we encourage the book's users to use the ArchStudio tool suite, as it has been developed for this purpose. The tool can benefit faculty, students, and even professionals using this text. The tool is free, is integrated with the Eclipse development environment, and has been used in the classroom and in industry. A community of developers and users has formed around this project, with dozens of projects around the world using and contributing. Further details can be found on the project's Web site: www.isr.uci.edu/projects/archstudio/.

Chapter Summaries

Chapter 1: The Big Idea

The chapter introduces software architecture through discussion of three primary examples of its application: in the very large (the architecture of the World Wide Web), in the very small (pipe-and-filter on the desktop), and in the product/in the many (the Philips/Koala architecture for consumer electronics).

Chapter 2: Architectures in Context: The Reorientation of Software Engineering

The role of software architecture in the major activities of software engineering is explored, including application conception, design, implementation, and analysis. An architecture-centric perspective on development is presented.

Chapter 3: Basic Concepts

Definitions of the key terms and concepts are presented. The definitions build from the examples in the preceding chapters and form the foundation for the subsequent chapters.

Chapter 4: Designing Architectures

The concepts of the preceding chapters are made productive in this one. This chapter presents, at length, techniques and approaches for designing applications from an architecture-centric perspective. The central focus is on the ways in which refined experience may be used to guide new developments.

Chapter 5: Connectors

Connectors play a distinguished role in architecture-centric development. They are also, perhaps, the most unfamiliar concept to those unskilled in the art of system design. This chapter reveals the breadth and depth of the concept, and includes an extensive guide to the numerous techniques available.

Chapter 6: Modeling

This chapter surveys an extensive range of modeling notations used for capturing the design decisions that make up architecture. Notations from research and practice are examined, and the distinctive aspects of each notation are called out. A consistent running example application is used throughout the chapter to demonstrate each notation.

Chapter 7: Visualization

This chapter draws a distinction between how architectures are modeled—the syntax and semantics of how design decisions are documented—from how they are visualized. Visualization encompasses both depiction—how design decisions are presented—and interaction—the “user interface” through which stakeholders interact with those depictions. Various visualization approaches are surveyed and compared, along with techniques for constructing and integrating new visualizations.

Chapter 8: Analysis

One of the primary benefits of software architecture is that it can enable early assessment of a given system, before significant resources are invested in the system’s implementation and deployment. This chapter provides a broad overview of the various analysis techniques that can be applied at the architectural level, their strengths, and limitations.

Chapter 9: Implementation

This chapter describes the problem of moving from architectural design decisions to implementation artifacts. Special attention is paid to the use of *architecture implementation frameworks*, special-purpose middleware that bridges the gap between architectural styles and existing platform services. Two complete implemented applications in alternative frameworks are presented with commentary.

Chapter 10: Deployment and Mobility

This chapter addresses deployment and mobility, two related topics that have become important in contemporary distributed, embedded, and pervasive systems. The objective of the chapter is not to provide a complete treatment of these two topics, but rather to isolate and focus on their characteristics from an explicit software architectural perspective.

Chapter 11: Applied Architectures and Styles

Several examples from “the practice” are explored, showing how various of the ideas in the preceding chapters have been applied to solve deep and commercially important problems. The architectures of the World Wide Web, Napster, Gnutella, Skype, BitTorrent, robotics systems, Google, and Akamai are discussed. The pedagogical goal is showing how multiple simple architectural approaches can effectively be combined.

Chapter 12: Designing for Non-Functional Properties

This chapter selects a set of non-functional properties that are frequently required of large and complex software systems: efficiency, complexity, scalability, adaptability, and dependability. The role of software architecture in achieving the desired level of each property is discussed, and many specific architectural design guidelines are provided and discussed.

Chapter 13: Security and Trust

Architecture-based techniques are presented for coping with security and trust needs in application system design. After a general introduction to security issues, attention is focused on architecture-based access control and reputation-based trust systems.

Chapter 14: Architectural Adaptation

The theme of this chapter is use of an architectural model to guide long-term system adaptation. A comprehensive conceptual framework for adaptation is presented, followed by consideration of numerous specific techniques for effecting adaptation and for designing applications to be accommodating of post-release adaptation.

Chapter 15: Domain-Specific Software Engineering

Architecture-centric development is perhaps most valuable when applied in the context of a specific domain, where reuse of hard-won engineering knowledge and principles can substantially reduce the amount of effort, cost, and risk involved in system development. This chapter surveys a variety of techniques for applying architecture-centric development within a domain, and ends with an extensive treatment of product-line architectures.

Chapter 16: Standards

A practicing architect or system developer is faced with a panoply of de facto and de jure standards—IEEE 1471, UML, SysML, DoDAF, RM-ODP, and so on. The chapter opens with a discussion of standards, how they are created and evolved, and how they can add value to development efforts. The remainder of the chapter surveys the most influential standards and attempts to put them in perspective, identifying their strengths and weaknesses explicitly.

Chapter 17: People, Roles, and Teams

An effective architect may need to possess many different skills, beyond just being a good software designer. An architect is an integral part of a larger organization, and may be one of several architects working on a project. The context within which software architects operate, and therefore their job description, can get quite complex. This chapter overviews the various roles software architects may play in a project and, more broadly, an organization.

Acknowledgments

The authors want to express their sincere appreciation to the large number of people who have been instrumental in bringing this text to fruition.

Hazel Asuncion was instrumental in the development of the various Lunar Lander examples, the domain model for the Lander, and was indefatigable as a reviewer. Girish Suryanarayana and Jie Ren were key contributors to the chapter on security and trust. Sam Malek and Marija Mikic-Rakic supplied several figures and a number of key insights in our discussions of deployment and mobility. Chris Mattmann provided the material for data-intensive connectors presented in Chapter 5. Joshua Garcia and Brian D'Souza contributed to the connector compatibility matrix in the same chapter. George Edwards provided the example approach used in illustrating scenario-driven analysis in Chapter 8. John Georgas provided essential material for the discussion of robotics architectures in Chapter 11.

Kari Nies was an especially helpful and careful reviewer of the entire manuscript. The students of UC Irvine's Fall 2006 and Fall 2007 graduate classes in software engineering (Informatics 211) and USC's Spring 2007 and Spring 2008 graduate classes in software architectures (CSci 578) were early reviewers and commentators on key parts of the manuscript; their help is gratefully acknowledged. The undergraduates in UC Irvine's Spring 2008 class in Software Architectures, Distributed Systems, and Interoperability also were helpful. Similarly, the comments of Peyman Oriezy, Justin Erenkrantz, Roy Fielding, Scott Hendrickson, Michael Gorlick, David Woollard, Roshanak Roshandel, Daniel Popescu, George Edwards, and Chris Mattmann were very much appreciated.

Dan Sayre of John Wiley and Sons was instrumental in shepherding three first-time authors through the harrowing and often confusing process of developing and writing a textbook. Our interactions with him convinced us that Wiley was the right place for this book. Our subsequent interactions with the editorial and production staff confirmed this positive experience. Heather Johnson and the copy editor provided pleasant and extremely helpful service, Lee Goldstein's book design skills were much appreciated and she was a pleasure to work with. Special thanks to Lyrica Taylor for recommending the painting by Thomas Cole for the cover and writing the explanatory text.

External reviewers included Professors George Heineman (Worcester Polytechnic Institute) and Anthony I. Wasserman (Carnegie-Mellon West), both of whom provided particularly helpful suggestions and recommendations. Professors André van der Hoek (UC Irvine), Jeff Kramer (Imperial College London), Hans van Vliet (Vrije Universiteit), Alexander Wolf (Imperial College London), LiGuo Huang (University of Kansas), Giuseppe Valetto (Drexel University), Tony Sullivan (University of Texas, Dallas), and John McGregor (Clemson University) also provided important guidance. Several anonymous reviewers also contributed key insights for the work, as did MIT Press's Robert Prior.

University College London deserves special thanks for hosting Taylor's sabbatical in 2005–2006, during which time the initial manuscript was substantially developed. A wonderful flat in Hampstead helped, too.

From the technical side, production of the book was dependent upon the services of Subversion for close to 1,000 commits of the manuscript. OmniGraffle was used for the majority of the diagrams; Skype kept us in touch while the authors were separated by thousands of miles.

The support of the National Science Foundation through grants CNS-0438996, CCF-0430066, CCF-0524033, CCR-9985441, ITR-0312780, and CSR-0509539, The Boeing Company, Bosch, the Jet Propulsion Laboratory, and IBM is gratefully acknowledged. Eric Dashofy would like to express his personal thanks to Dr. Michael and Julie Penley and the ARCS Foundation, Orange County Chapter, for their generous support.

—Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy

About the Authors

Richard N. Taylor is a Professor of Information and Computer Sciences at the University of California, Irvine, in the Department of Informatics. He received a Ph.D. degree in computer science from the University of Colorado at Boulder in 1980. Professor Taylor is Director of the Institute for Software Research, which is dedicated to fostering innovative basic and applied research in software and information technologies through partnerships with industry and government. He has taught courses at the graduate and undergraduate levels in software engineering, software design, project development, and software engineering environments. He is the author of more than 100 journal and conference publications and has chaired the dissertation committees of over twenty Ph.D. graduates. He has served as chairman of ACM's Special Interest Group on Software Engineering, SIGSOFT, and as chairman of the steering committee for the International Conference on Software Engineering (ICSE). He has served as an expert witness in several intellectual property disputes. Taylor was a 1985 recipient of a Presidential Young Investigator Award. In 1998, he was recognized as an ACM Fellow and in 2005 was awarded the ACM SIGSOFT Distinguished Service Award.

Nenad Medvidović is an Associate Professor in the Computer Science Department at the University of Southern California. He received his Ph.D. in 1999 from the Department of Information and Computer Science at the University of California, Irvine. As of January 1, 2009, he is the Director of the USC Center for Systems and Software Engineering (CSSE). Medvidović has taught undergraduate, graduate, and industry courses in software engineering, software architecture, software engineering for embedded systems, and formal methods. He has published more than 100 journal and conference papers and has chaired the dissertation committees of seven Ph.D. graduates. Along with Richard N. Taylor and Peyman Oreizy, he co-authored the ICSE 1998 paper titled, "Architecture-Based Runtime Software Evolution," which in 2008 was named that conference's most influential paper. Professors Medvidović and Taylor's paper, "A Classification and Comparison Framework for Software Architecture Description Languages," was recognized by the Elsevier Information and Software Technology Journal as the most cited journal article in software engineering published in 2000. Medvidović served as a consultant to NASA's Jet Propulsion Laboratory in creating their software architect training program (SWAP). He is a recipient of the National Science Foundation CAREER award, the Okawa Foundation Research Grant, and the IBM Real-Time Innovation Award.

Eric M. Dashofy is a senior member of the technical staff in the Computer Systems Research Department at The Aerospace Corporation in El Segundo, CA. He received his Ph.D. in 2007 from the Donald Bren School of Information and Computer Sciences at the University of California, Irvine. He has written numerous technical publications in software engineering and architecture. He is the lead developer of the ArchStudio architecture-centric software development environment, an open-source, Eclipse-based tool set for modeling, visualizing, analyzing, and implementing architecture-based software systems. He also co-authored the xADL 2.0 architecture description language, upon which ArchStudio is based. His current work focuses on software architecture, high-performance computing, and wireless sensor networks of embedded devices.

The Big Idea

The study of software architecture is the study of how software systems are designed and built. A system's architecture is the set of principal design decisions made during its development and any subsequent evolution. As a subject, architecture is the proper primary focus of software engineering, for the production of high-quality, successful products is dependent upon those principal decisions.

An architecture-centric approach to software development places an emphasis on *design* that pervades the activity from the very beginning. Design quality correlates well with software quality—it would be extremely unusual to find a high-quality software system with a poor design. The practice of architecture-centric development can also enable the creation of cost-effective families of software products that can dominate an application sector over an extended period of time. Good design practices can leverage the lessons of experience and provide strategies for effectively meeting a wide range of needs.

This chapter introduces the central ideas of software architecture. It does so by first exploring the analogy between the architecture of buildings and the architecture of software. Software architecture, and the power that comes from making it the centerpiece of system development, is then illustrated three ways. First, the architectural ideas underpinning the World Wide Web are explored—software architecture “in the very large.” Second, the ideas are explored on the desktop—software architecture in the small. Third, the central role of architectures in enabling successful product families is explored, using consumer electronics as the application domain. We deliberately omit most technical details, seeking rather to instill a sense of “the big idea.”

**Outline of
Chapter 1**

1 The Big Idea	1.4 The Power of Architecture in Business: Productivity and Product Lines
1.1 The Power of Analogy: The Architecture of Buildings	1.5 End Matter
1.1.1 Limitations of the Analogy	1.6 Review Questions
1.1.2 So, What's the Big Idea?	1.7 Exercises
1.2 The Power and Necessity of Big Ideas: The Architecture of the Web	1.8 Further Reading
1.3 The Power of Architecture in the Small: Architecture on the Desktop	

1.1 THE POWER OF ANALOGY: THE ARCHITECTURE OF BUILDINGS

The discipline of architecture, that is, the design and construction of buildings, offers a rich base of concepts from which software architecture, and more generally, software engineering has drawn. The analogy between the design and construction of buildings and the design and construction of software is strong and readily apprehended, since we all have substantial experience in living in and around buildings and in seeing them built.

Software engineering textbooks typically use this analogy to motivate the phases of the traditional software life cycle. In a highly simplified and idealized conception of architecture, requirements for a building are collected, a design is created to satisfy those requirements, the design is refined to yield elaborate blueprints, construction is based on the blueprints, and the resulting structure is then occupied and used. So, notionally, in the software domain, requirements are specified, a high-level design is created, detailed algorithms are developed based upon that design, code is written to implement the algorithms, and finally the system is deployed and used.

The idealized summary above of how buildings come to be is almost trivial, but offers several insights reflected in the software domain. For instance, the architectural process has as its focus the satisfaction of the future occupant's needs. It allows for specialization of labor: The designer of the structure need not be the contractor who performs the actual construction. The process has many intermediate points where plans and progress may be reviewed. Thus there is the corresponding simplistic view of software development: Specification of a system's requirements precedes its design; that design is created by specialists, not by the ultimate users of a system. Actual programming may be contracted out, even sent offshore. Prototypes and mock-ups created at various points during development enable the customer to periodically assess whether the emerging system will indeed meet identified needs.

A more serious consideration of architecture, however, reveals deeper insights. First and perhaps foremost is the very conception of a building having an architecture, where that architecture is a concept separate from, but inextricably linked to, the physical structure itself. A building's architecture, that is, its major elements, their composition, and arrangement, can be described, discussed, and compared with those of other buildings. The architecture that was in the mind of the architect early in the development process can be compared with the architecture of whatever physical structure emerged from the construction process. So, too, as we will shortly describe, the principal design decisions

characterizing a software application—that is, its architecture—exist independently from, but linked to, the code that ostensibly implements that architecture.

In the first century AD, the Roman author Marcus Vitruvius Pollio, best known as Vitruvius, produced a famous treatise on architecture, entitled *De Architectura*. This handbook contains numerous admonitions, opinions, and observations about architecture and architects that remain, two thousand years later, insightful and worth considering. He begins his treatise in a remarkable place, considering the practice and the theory of architecture. Both are necessary for the professional. Consider his words:

Practice and Theory

Practice is the frequent and continued contemplation of the mode of executing any given work, or of the mere operation of the hands, for the conversion of the material in the best and readiest way. Theory is the result of that reasoning which demonstrates and explains that the material wrought has been so converted as to answer the end proposed. Vitruvius I, 1, 1

(All Vitruvius quotes in this text are from Pollio)

Note that practice is not the mere “doing,” but involves contemplation of what is done.

A second insight is that properties of structures are induced by the design of their architectures. For instance, a medieval castle with high, thick walls and narrow or nonexistent windows is designed that way so that it has excellent defensive properties (as long as attackers are armed only with swords and arrows). So, too, as we will see, properties of software applications, such as resilience in the face of particular types of security attacks, are determined by the design of their architectures.

All these should possess strength, utility, and beauty. Strength arises from carrying down the foundations to a good solid bottom, and from making a proper choice of materials without parsimony. Utility arises from a judicious distribution of the parts, so that their purposes be duly answered, and that each have its proper situation. Beauty is produced by the pleasing appearance and good taste of the whole, and by the dimensions of all the parts being duly proportioned to each other. Vitruvius I, 3, 2

Strength, Utility, and Beauty: The Qualities Obtained from Good Architecture

A third insight is recognition of the distinctive role and character of an architect, the person responsible for the creation of the architecture. The discipline of architecture has long recognized that architects require very broad training. While competence in aspects of engineering is necessary, much more than that is required. A fine sense of aesthetics and a deep understanding of how people work, play, eat, and live are essential in creating buildings that are enjoyed, satisfy their occupants, and perform effectively over the seasons and through the years. In a like manner, simple skill in programming is not sufficient for the creation of complex software applications that people can effectively employ.

An architect should be ingenious, and apt in the acquisition of knowledge. Deficient in either of these qualities, he cannot be a perfect master. He should be a good writer, a skillful draftsman, versed in geometry and optics, expert at figures, acquainted with history, informed on the principles of natural and moral philosophy, somewhat of a musician, not ignorant of the sciences

Vitruvius on Qualifications of an Architect

both of law and physic, nor of the motions, laws, and relations to each other, of the heavenly bodies. Vitruvius, 1, 1, 3

A fourth insight is that process is not as important as architecture. This is not to say that process is unimportant. On the contrary, architects and construction companies clearly follow and depend upon standard processes to guide their daily activities and ensure that all aspects of the design and build activities are addressed. But there is never any question that the product—the architecture—is the central focus. Simply following a standard process will not guarantee that a successful building will emerge, meeting the needs of its owners and occupants. The architects and engineers responsible for the structure must keep its design and qualities at the forefront; process is present to serve those ends, not to be an end in itself.

A fifth insight is that architecture has matured over the years into a discipline: A body of knowledge exists about how to create a wide range of buildings to meet many types of needs. It is not simply a discipline of basic principles and generic development processes, however. If every building had to be designed from first principles and the properties of materials had to be rediscovered for each new project, then most of us would be wet and shivering with the sky for our roof.

The discipline of architectural engineering has captured the experiences and lessons of previous generations so that the process of designing, for instance, a new suburban home is much like the process of designing a thousand other homes. This is not to say that the homes are identical; rather, within the broad concept of “suburban home” some basics are established and points of allowable variation are known. Where there is commonality between two homes, great efficiencies can be realized through reuse of knowledge, reuse of subsystem design, reuse of tools, and the benefits that come from standardized materials, parts, and sizes. While anyone who has ever endeavored to build a custom home would certainly dispute that the process is efficient compared to the activity of designing from a truly clean slate, the craft works very well. So, too, as the ensuing chapters of this book will demonstrate, software architecture is quickly maturing into a robust discipline, leveraging the knowledge gained through a myriad of system development experiences.

One fundamental way in which the experiences and lessons from previous generations of architects and building-dwellers has been captured is summed up in the notion of *architectural styles*—an insight that has powerful application in the domain of software. The phrases “Roman villa,” “Gothic cathedral,” “ranch-style tract home,” “Swiss chalet,” and “New York skyscraper” each characterize types of buildings that have various features in common. Ranch-style tract homes, for instance, are single-story residences with low roofs; Swiss chalets are usually two to four stories, traditionally made of timber, have steep roofs, and large sheltered balconies. New York skyscrapers have many dozens of stories, have steel frames, make superb use of small footprint building areas, and offer hundreds of thousands of square meters of floor space. Roman villas suit a Mediterranean climate, and so on.

The development of an architectural style over time reflects the knowledge and experience gained by the builders and occupants as they try to meet a common set of requirements and accommodate the constraints of the local topography, weather, available building materials, tools, and labor. Ranch-style houses, such as those prevalent in Southern California, can be inexpensively built if lumber for framing is readily available, function very well in earthquake-prone areas, and are excellent for individuals who cannot climb stairs. Swiss

chalets work well in areas with heavy snowfall, with the steep roofs assisting in minimizing the structural load caused by the snow. Therefore an approach that works well for providing homes in Southern California is not necessarily going to be appropriate for providing homes in Switzerland, or vice versa. But within Southern California a wide variety of site-suitable houses can be successfully built in a cost-effective manner by those architects skilled in the local idioms and materials.

[Private buildings] are properly designed, when due regard is had to the country and climate in which they are erected. For the method of building which is suited to Egypt would be very improper in Spain, and that in use in Pontus would be absurd at Rome: so in other parts of the world a style suitable to one climate, would be very unsuitable to another: for one part of the world is under the sun's course, another is distant from it, and another, between the two, is temperate. Vitruvius, VI, 1, 1

Vitruvius on Styles

One way architectural styles can be summed up is as a set of constraints—constraints put upon development in order to elicit particular desirable qualities. For example, the Swiss chalet style has a constraint that the roofs have steep slopes; the quality elicited is that chalets tend to do well in areas of heavy snowfall. The suburban ranch style has a constraint that commodity components be used; the quality being that the houses are cheaper to build and easier to fix than custom homes. The Gothic style constrains one to building with stone, stained glass, and high fluted vaults; the qualities elicited are that the buildings are long-lived, instructional, and inspirational.

Characterized in more constructive terms, styles offer the architect a wide range of solutions, techniques, and palettes of compatible materials, colors, and sizes. Rather than spending a large amount of time searching through an unbounded space of alternatives, by working within a style an architect can spend that same amount of time refining, customizing, and perfecting a particular design.

The concept of architectural styles carries over very powerfully into the domain of software. As we will illustrate in the remainder of this chapter and then throughout the book, styles are essential tools for architects to master, for they are a major point of intellectual leverage in the task of creating complex software systems.

1.1.1 Limitations of the Analogy

Before pressing on to the detailed consideration of architectures, styles, and their use in software systems development, a few cautionary words are in order with regard to the use of the analogy to building architectures. As with any analogy, it has limitations.

First, we know a lot about buildings. That is, since birth we all have experienced and learned about buildings. As a result, we have a well-developed intuition as to what kinds of buildings can and cannot be built, and what is appropriate for a given need and situation. Our intuitions for software are not nearly so well-developed and hence we must be more methodical and more analytical in our approach.

Second, the essential nature of the software medium is fundamentally different from the materials and media of building architecture. You can discern much of a building's architecture just by looking at it. With software the problem is much more difficult. Software

is intrinsically intangible; at core it is an abstract entity and we only work with various representations of it. This implies that software is more difficult to measure and analyze, making it more difficult to evaluate the various qualities of designs and measure progress towards completion.

Third, software is more malleable than physical building materials, offering the possibility of types of change unthinkable in a physical domain. The building analogy is thus a poor source of ideas for dealing with change, since buildings accommodate change with difficulty¹.

There are additional problems with the analogy:

- There is no software construction industry in the same degree that there is for buildings. The building industry has substantial substructure, reflecting numerous specializations, skill sets, training paths, corporate organizations, standards bodies, and regulations. While the software industry has some structure, including that seen in offshore development practices, it is much less differentiated than the building industry.
- The discipline of architecture does not have anything akin to the issue of deployment as found in software: Software is built one place, but deployed for use in many places, often with specialization and localization. Manufactured buildings, otherwise known as trailers, are somewhat similar, but still there is no corresponding notion to dynamic distributed, mobile architectures, as there is with software.
- Software is a machine; buildings are not (notwithstanding Le Corbusier's declaration that, "A house is a machine for living in"). The dynamic character of software—the observation that led to Edsger Dijkstra's famous "goto statement considered harmful" paper (Dijkstra 1968), provides a profoundly difficult challenge to designers, for which there is no counterpart in building design.

Despite these limitations—and others—the analogy between the architecture of buildings and software is strong and instructive. The focus on architecture is critical in the design of buildings; such a focus is similarly powerful for software. Subsequent sections of this chapter will demonstrate this in the large, in the small, and in the crucible of industry. Before considering these examples, however, we summarize our main themes in the next section.

1.1.2 So, What's the Big Idea?

The big idea is that software architecture must be at the very heart of software systems design and development. It must be in the foreground, more than process, more than analysis, and certainly more than programming. Only by giving adequate attention and prominence to the architecture of a software system, over its entire lifespan, can that system's development and long-term evolution be effective or efficient in any meaningful sense. Indeed, we will see that for any application of significant size or complexity, its architecture must be considered in advance, just as the successful creation of a large building requires consideration of its architecture in advance of construction.

¹This topic will be explored in more detail in Chapter 14. See also (Brand, 1994).

Furthermore, the insights presented above about architecture, architects, and especially, architectural styles offer substantial intellectual leverage in the creation of software systems, but demand careful study, good tools, and disciplined use to yield their substantial potential benefits.

Giving preeminence to architecture offers the potential for realizing:

- Intellectual control
- Conceptual integrity
- An adequate and effective basis for reuse—of knowledge, experience, designs, and code
- Effective project communication
- Management of a set of related variant systems

Note the phrase above about directing “attention and prominence to the architecture of a software system, *over its entire lifespan*.” A limited-term focus on software architecture will not yield significant benefits. Just as the concept of architecture, or the involvement of an architect, in a building project is no guarantee that a successful building will be created, so it is for software. Shortcomings in the construction process—whether of buildings or of software—can cause the built system to deviate from its intended architecture, possibly with disastrous consequences. Such shortcomings can be avoided, however, and we will discuss techniques specifically intended to ensure that the implemented system is, and remains, faithful to its intended architecture.

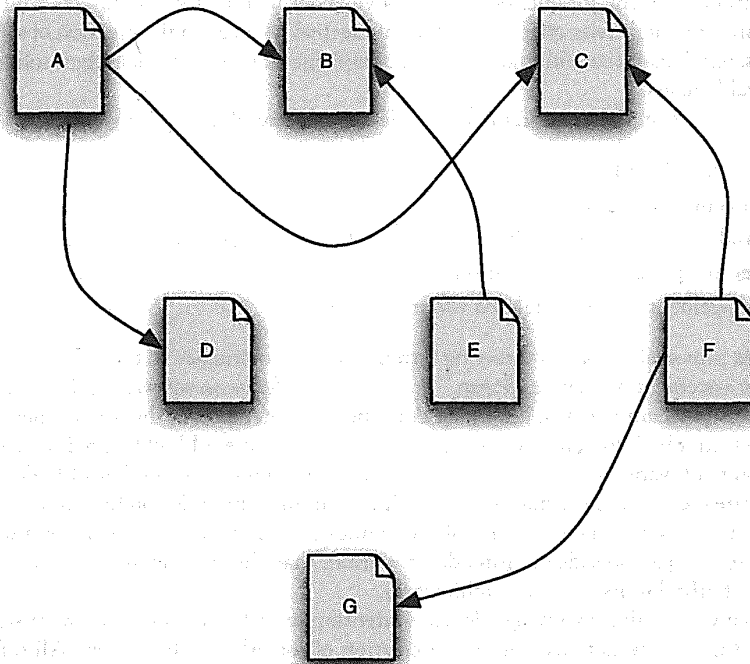
Finally, note that by saying adequate attention must be given to a software system’s architecture we are not advocating the creation of something wholly new: All software systems have an architecture. Just as every building has an architecture and at least one architect, so does software. Simply stating that a building or a program has an architecture or an architect does not imply much. There are good architectures, bad architectures, elegant ones, and curious ones. So it is with software. An application’s structure may be elegant and effective or clumsy and dysfunctional. Our objective in the coming pages is to give the reader the skills necessary to ensure that applications have good, elegant, and effective architectures. The following sections proceed by considering some outstanding examples of the discipline of software architecture, well applied.

1.2 THE POWER AND NECESSITY OF BIG IDEAS: THE ARCHITECTURE OF THE WEB

A primary example of the power of architecture can be seen in an application all readers of this book are familiar with: the World Wide Web. Think for a moment: What is the Web? How is it built? How do you explain the Web to a child? If you have a business and want to have a Web-based e-commerce presence, how do you go about designing the software for your site, including determining how your site will interact with your customers’ machines? It is *architecture* that offers the vocabulary and the means for answering these questions. It is the particular *architectural style* of the Web that constrains (and thereby helps) you in producing an e-commerce system that “plays well” with others.

Let’s answer some of the questions above. What is the Web? In one view, focusing on the user’s perception, the Web is a dynamic set of relationships among collections of

Figure 1-1.
A document
hypertext.



information. In another view, focusing on coarse-grained aspects of the Web's structure, the Web is a dynamic collection of independently owned and operated machines, located around the world, which interact across computer networks. In another view, taking the perspective of an application developer, it is a collection of independently written programs that interact with each other according to rules specified in the HTTP, URI, MIME, and HTML standards.

Considering these perspectives in turn, the view of the Web as a collection of interrelated pieces of information is illustrated in Figure 1-1.

In the figure documents labeled A to G are shown as independent entities, but with explicit relationships among them. For instance, if document A is a biography of the author C. S. Lewis, then the arrow to document D might represent a view of *The Kilns*, the house in Oxford where Lewis lived for many years, as imaged by a webcam. The arrow to document C from document A might represent a reference to a description of Oxfordshire, the county in which *The Kilns* is found. The other documents shown might represent the text of some of the many books that Lewis wrote, such as *The Lion, the Witch and the Wardrobe*. We, as users of the Web, can understand Figure 1-1's mini-Web as set of interrelated documents and a Web browser, such as Safari or Internet Explorer, as a vehicle for viewing the documents and navigating among them. In short, this mini-Web is known as a *hypertext* and the viewing and shifting of focus from one document to the next as browsing, or surfing, this hypertext.

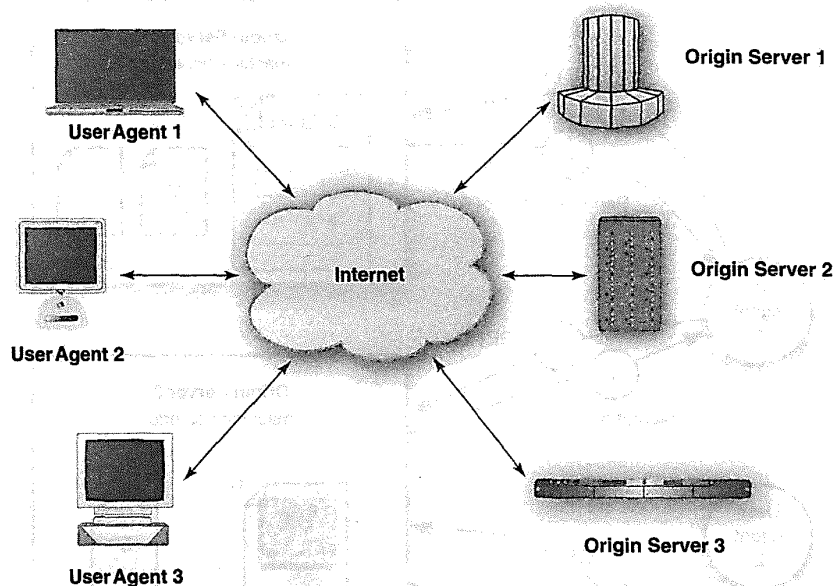
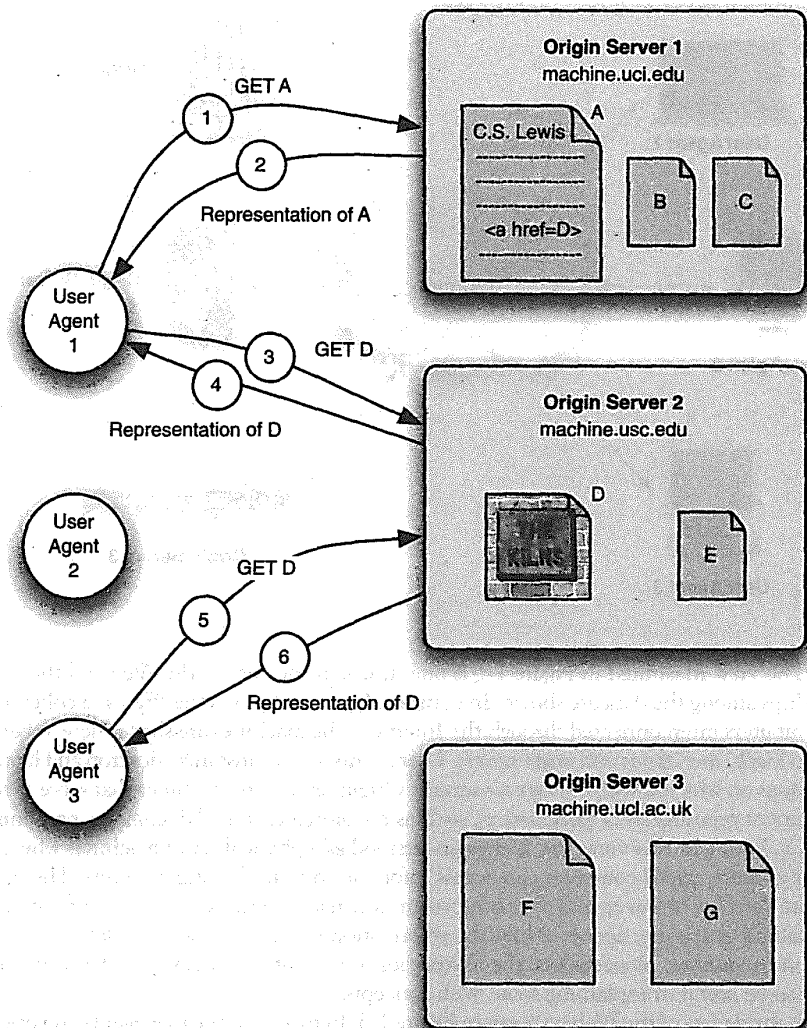


Figure 1-2.
A machine view
of a small part of
the World Wide
Web.

The view illustrated in Figure 1-1 is one in which the data of the Web and the relationships among the data are shown. In contrast, Figure 1-2 shows the Web as a collection of computers interconnected through the Internet. The machines are shown here as being of two kinds: *user agents* and *origin servers*. User agents are, for instance, desktop and laptop machines on which a Web browser is running. Origin servers are machines that serve as the permanent repositories of information, such as the aforementioned documents pertaining to C. S. Lewis. In this view, the Web is understood as a physical set of machines whereby a user at a user agent computer can access information from the origin servers. The view is quite sketchy, however, and does not present any real insight into how the information is obtained by the user agents or how the information in one document is related to information in another. Nonetheless, the abstraction it presents is accurate insofar as it goes, and can be useful in explaining some Web concepts.

A third view of the Web is shown in Figure 1-3. In this view, the user agents and origin servers of Figure 1-2 are once again shown, but now the location of documents A to G are shown. The figure also shows a set of specific interactions among two of the user agents and two of the origin servers, corresponding to a particular pattern of accessing the C. S. Lewis hypertext. In this example, User Agent 1 requests, by means of the HTTP method GET, a copy of document A, a biography of Lewis. This interaction is shown in the diagram by the arrow marked 1. A representation, or "copy," of the biography is returned, shown as the arrow marked 2. Since the biography has within it a hypertext reference (href) to document D, a JPEG of The Kilns, User Agent 1 issues another GET, this time to machine.usc.edu, to obtain the picture. This request is marked 3 in the diagram. A copy of the image is returned, as shown by the arrow marked 4. Further interactions between user agents and servers are shown in the diagram as interactions 5 and 6.

Figure 1-3.
Agents and origin
servers
interacting
according to the
HTTP protocol,
as users peruse
the Web.



These three diagrams illustrate the Web and provide an indication of how it works, but clearly there is much, much more to it. These diagrams use a very simple, limited set of information to represent the billions of pages of information available on the Web, and a set of less than a dozen machines to represent the millions of machines interacting at any given instant over the Internet. So, can we really say that these diagrams explain how the Web works? Clearly not. A much more general understanding of the Web can be given as a set of definitions and constraints on how these billions of documents interrelate and

millions of machines interact,² as follows:

- The Web is a collection of *resources*, each of which can be identified by a uniform resource locator, or URL. A standard specifies the legal syntax of a URL.
- Each resource denotes, informally, some information. The information may be, for example, a document, an image, a time-varying service (for example, “today’s weather in Los Angeles”), a collection of other resources, and so on.
- URLs can be used to determine the identity of a machine on the Internet, known as an *origin server*, where the value of the resource may be ascertained.
- Communication is initiated by clients, known as *user agents* who make requests of servers. Web browsers are common instances of user agents.
- Resources can be manipulated through their *representations*. For instance, a resource may be updated by a user agent sending a new representation of that resource to the origin server that holds that information. Similarly a resource may be viewed by a user agent obtaining a representation of that resource from an origin server and displaying that representation on a monitor. HTML is a very common representation language used on the Web.
- All communication among user agents and origin servers must be performed in accordance with a simple, common protocol (HTTP). Communicating according to HTTP requires that the parties implement a few primitive operations, such as GET and POST.
- All communication between user agents and origin servers must be context-free. That is, an origin server must be able to respond correctly to a user agent’s request based solely on information contained in the request, and not require maintenance of a history of interactions between that user agent and the origin server. (This is sometimes known as “stateless interactions.”)

To illustrate, in the example above, documents A to G are resources; machines machine.uci.edu and machine.usc.edu running the Apache Web server are example origin servers. User agents could be personal laptops running a Web browser such as Internet Explorer. Representations include a copy of the HTML of Lewis’s biography, and a JPEG of the current view of The Kilns as imaged by a webcam.

Describing the rules by which the various parts of the Web work and interact—its *architectural style*—provides the basis for understanding the Web independent of its configuration or actions at any particular instant. Such description enables us to effectively reason about how the Web works and guides in determining what must be done to incorporate new information or new machines into the Web. While the list above is not complete (more details are provided later in Chapter 11 and in various references) it is nonetheless representative and substantive. This approach to understanding the Web is clearly superior to one based upon the code—trying to state every detail of every machine and every piece of software engaged in the Web at one particular time.

A number of critical observations are apparent:

- The architecture of the Web is wholly separate from the code that implements its various elements. Indeed, to understand the Web, the architecture is the *only* effective

²This characterization still omits many details! A more comprehensive description appears in Chapter 11.

reference point. The architecture is the set of principal design decisions that determine the key elements of the Web and their interrelationships. These decisions are at an abstraction level above that of the source code, and are thus conducive to understanding the entire system at the application level.

- There is no single piece of code that “implements the architecture.” The Web is “implemented” by Web servers of various design, browsers of various design, proxies, routers, and network infrastructure. Just looking at any single piece of code, or even all the code on any single machine, will not explain the Web’s structure; rather, the architecture is the only adequate guide to understanding the whole.
- There are multiple equivalent (with respect to the architecture) pieces of code that exist and implement the various components of the architecture. The architectural style of the Web constrains the code in some respects, saying how a given piece must work with respect to the other elements of the architecture, but substantial freedom for coding the internals of an element is present. Thus, we see many different browsers from different vendors, offering a variety of individual advantages, but insofar as the browsers relate to the rest of the Web, they are equivalent.
- The stylistic constraints that constitute the definition of the Web’s style are not necessarily apparent in the code, but the effects of the constraints as implemented in all the components are evident in the Web.

These observations are profound, and begin to indicate the role of architecture in the Web. But the most important questions of all remain:

- Why were these particular decisions made?
- Why were these decisions important and not others?
- Why did similar systems that made slightly different decisions fail, when the Web is such a wild success?

These questions can be answered only when looking at the Web from an architectural perspective, and indeed they cut straight to the heart of architecture. Chapter 11 discusses how the Web’s designers targeted a particular set of qualities—the ones that make the Web so successful—and then made decisions specifically to imbue the Web with those qualities. That chapter will discuss the Web and its underlying style, REST (REpresentational State Transfer), in more detail, and show how the style is based upon and derivative from a large set of simpler styles.

The take-away message here is that one of the world’s most successful software systems is understood adequately only from an architectural vantage point. The development of the Web, the maturation of the HTTP/1.1 protocol, and the implementation of its core elements were all driven by architectural understandings and principles. Without the rock of this abstraction it is unlikely the Web would have survived past its first two or three years of existence.

1.3 THE POWER OF ARCHITECTURE IN THE SMALL: ARCHITECTURE ON THE DESKTOP

One need not look at large complex applications, such as the Web, to find interesting architectures or to find applications where a focus on architectures has a big payoff.

Architectures underlie the simplest applications, and architectural concepts provide the conceptual power behind, for example, the nearly ubiquitous command-line shell programs. Found on virtually every platform, including Mac OS X, Linux, Windows, and the Unix platforms where the concepts originated, such scripts enable the user to quickly and easily compose new applications from preexisting components called filters (which happen to be complete executable programs in their own right), just by following some simple rules. For example, the following application creates a sorted list of all the files in the directory named `invoices` whose names include the character string “August”:

```
ls invoices | grep --e August | sort
```

At first blush this may not seem to be an application, since we can visually discern how the functionality is provided from piece-parts, but that is indeed what it is. To understand how this application works and how it is built, one must understand in general what filters and pipes are, understand what the specific filters used in building this application do, and finally, reason about how these particular filters are configured, using the pipes, to form this application.

First, a *filter* is a program that takes a stream of text characters as its input and produces a stream of characters as its output. A filter's processing may be controlled by a set of parameters. A *pipe* is a way of connecting two filters, in which the character stream output from the first filter is routed to the character stream input of the second filter.

In the application above, three filter programs are used: `ls`, `grep`, and `sort`. The `sort` filter examines its input stream, noting how the stream is divided into lines of text by an end-of-line character, and produces on its output stream those same lines of text, but in sorted order. While `sort` may be given optional parameters to control, for instance, whether the lines are sorted in ascending or descending order, in the application above no parameters are provided, so the default behavior is to sort in ascending order.

The `grep` filter examines its input character stream for lines (that is, portions of the input character stream demarcated by end-of-line characters) that contain a substring matching a string value provided as a parameter. In the application above, the `--e` parameter is used to provide the string value (“August”) for which `grep` is to search. `grep` produces as its output stream only those lines that contain the designated search string.

The `ls` filter does not process an incoming stream of characters; instead it communicates with the operating system to obtain the names of files in a named directory (or “folder”). `ls` then produces, as its output stream, a sequence of characters that are the textual names of the files in the directory designated by the command-line parameters to `ls`, with each file name followed by an end-of-line character.

With this understanding of filters as programs that read and produce character streams, pipes as ways of hooking up filters by routing the character streams, and the functioning of each of the three filters used (listing file names, looking for the presence of a particular substring, and sorting), it is easy to understand how the application above works. `ls` produces a textual list of files found in the `invoices` directory. A pipe (shown on the command line as a vertical bar) routes this output to the input of `grep`. `grep` then examines those names to identify any containing the substring “August,” and produces only those names as its output stream. A second pipe routes that output stream on to `sort`, which then sorts the file names in ascending order, producing that ordered listing as its (and the application's) output.

The critical observation here is that this application's structure can be understood on the basis of a very few rules. Given understanding of that structure and knowledge of the functioning of the individual filters, the function of the complete application can be understood. Knowledge of those same rules and of the functioning of a few dozen, preexisting, basic filter programs allows one to understand hundreds of other useful applications. Similarly, a developer may readily create new applications based upon those same rules and knowledge of the filters.

Common Unix Filter Programs

These commonly used Unix filter programs can be used individually or combined into pipe-and-filter architectures to create many useful applications. Consult your system's manual pages for detailed instructions on how to use them (try `'man man'` in a terminal window).

ls: List the names of files within a directory.

grep: Produce lines on output that match a specified pattern, where that pattern is described as a regular expression.

sort: Sort all input lines according to a set of parameters (for example, for ascending or descending order).

cat: Concatenate files specified by parameters to the output stream.

sed: Read the input stream, modify (edit) it according to specified parameters, and write the result to the output.

awk: Match patterns in the input stream and perform specified operations upon a match.

head: List the first *n* lines of the input on the output.

tail: List the last *n* lines of the input.

uniq: Copy unique input lines to the output.

less: Incrementally list the input on the output, with controls determining how much information to list at a given time.

lpr: Send input to the printer.

cut: Select portions of each input line and write them to the output.

man: Format and display the on-line manual pages for a specified command to the output.

tee: Copy the input to the output, plus make a copy in zero or more specified files.

A complex example:

```
grep HTTP_USER_AGENT httpd.state.log | cut -f 2- |
grep -v 'via Gateway' | grep -v 'via proxy gateway'
| sort | uniq -c | sort -nr | head -5
```

This produces the top five browsers (user agents) from a WWW state log called "httpd.state.log" eliminating all proxy user agent strings.

The particular set of rules at work here defines an architectural style known, not surprisingly, as pipe-and-filter. Part of the beauty of pipe-and-filter is that because of its simplicity end users can develop applications without ever being trained as programmers. It is akin to working with Lego blocks, or Tinker Toys: Once you understand how to fit the parts together and the different kinds of functions that the various piece-parts are capable of performing, the creative task of assembling the pieces into a new design can proceed quickly and effectively. Training in the details of programming is not required; the power of a simple architectural concept can be comprehended and applied by a broad audience.

Though the style is very simple, note that use of it is not confined to the use of the standard filter programs found in Unix, Linux, and the other operating systems that

support pipe-and-filter; a developer may create a program that operates by reading and writing a character stream and then use that filter/program in conjunction with others in a pipe-and-filter-based application. Similarly, the style is not confined to applications written in the command-line notation, though that notation is certainly common.

In addition to pipe-and-filter, a wide variety of other simple architectural styles exist. Most of these will be familiar to experienced developers: layered system, main program and subroutines, object-oriented, implicit invocation, and blackboard. A detailed discussion appears in Chapter 4. Developers can choose among these styles, and a vast array of others, based upon the nature of the problem to be solved and the qualities desired in the solution. For instance, pipe-and-filter applications are readily understandable, run on almost any operating system, and run efficiently when the problem (and its structuring in pipe-and-filter) admits concurrency. On the other hand, pipe-and-filter requires all communication between filters to be serialized into character streams; thus, if a graph data structure has to be passed from one filter to another it must first be serialized into a textual stream, transferred across the pipe, and then rebuilt into the graph data structure by the next filter. The pipes ensure syntactic compatibility between the filters, but do nothing to ensure semantic compatibility.

1.4 THE POWER OF ARCHITECTURE IN BUSINESS: PRODUCTIVITY AND PRODUCT LINES

The discussion of the World Wide Web in Section 1.2 illustrated how architecture is a critical enabler for the development of large-scale, complex systems. The discussion of the pipe-and-filter style in Section 1.3 showed how architectural concepts can make effective the development of even online applications, providing the leverage needed for exploiting the power of a library of reusable components (in particular, Unix filters). Architectures are also critical enablers for developing *product families*, a key element of many business strategies.

Product families are sets of independent programs that have a significant measure of commonality in their constituent components and structure. The key concepts are shown in the following example. Any purchaser of consumer electronics, such as televisions, is aware that myriad choices are available. Even from a single manufacturer, the consumer is faced with a range of sizes and features that allows one to purchase a device meeting a very particular set of requirements. Perhaps a purchaser wants a 35-inch HDTV with a built-in DVD player for the North American market. Such a device might contain upwards of a million lines of embedded software. This particular television/DVD player will be very similar to a 35-inch HDTV without the DVD player, and also to a 35-inch HDTV with a built-in DVD player for the European market, where the TV must be able to handle DVB-T broadcasts, rather than North America's ATSC format. Each of these closely related televisions will have a million or more lines of code embedded within them.

The economic challenge from the manufacturer's point of view is to produce the wide range of products that a worldwide market of sophisticated consumers demands while simultaneously exploiting the commonalities among members of a product family. Reusing structure, behaviors, and component implementations is increasingly important to successful business practice because:

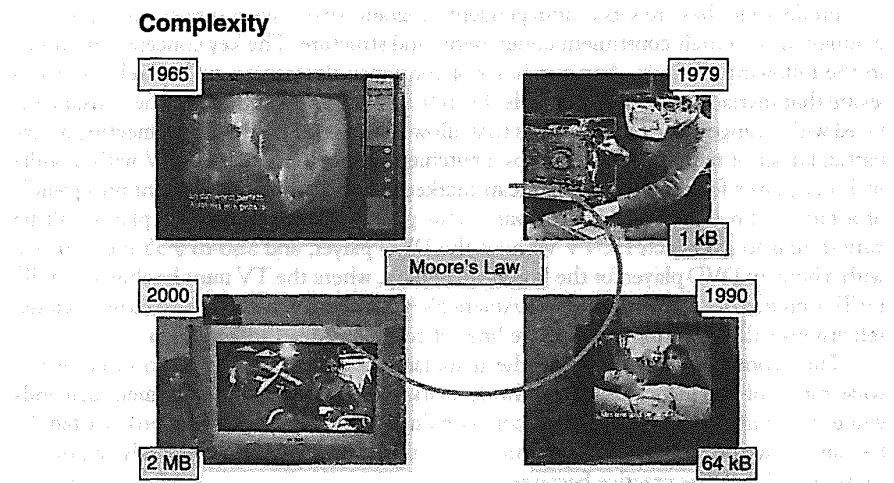
- It simplifies the software development task: Existing design- and implementation-level solutions can be either directly applied or easily adapted to multiple products within a family.
- It reduces the development time and cost: Part of the functionality needed for a new product will already exist within previous products within the same family.
- It improves the overall system reliability: Any functionality that is reused from a previous product will have been used and tested more extensively than if developed anew.

Software architecture provides the critical abstractions that enable variation and commonality within a product family to be simultaneously managed. We will provide an extensive treatment of this subject in Chapter 15. Here, we discuss a representative example that demonstrates the power of product families and the benefits accrued via an explicit, and extensive, software architectural focus.

The business case for exploiting software architecture for this task has been recognized by one of the world's leading consumer electronics manufacturers, Philips. Since the late 1990s, Philips progressively has developed and applied their Koala technology for specifying and implementing the architectures of their mid- and high-end television sets, and has hundreds of software engineers exploiting the concepts.

The motivation for Koala came directly from the nature of and advances in the consumer electronics domain, as shown in the case of Philips television sets in Figure 1-4. While early televisions supported a very small number of simple functions, over time they became more powerful with increasingly sophisticated hardware and, eventually, software capabilities. The resulting growth in complexity carried with it the risks of ever-increasing costs and lengthened time-to-market. Of course, this is precisely the opposite of what modern consumers have come to expect: Fierce competition has forced Philips, as well as its rivals, to produce a steady stream of new products and variations on existing products, while containing their costs.

Figure 1-4.
More sophisticated products carry with them increased complexity, which in turn implies increased cost and worse time-to-market unless the problem is addressed.
(Figure courtesy Royal Phillips N.V.)



A Television Product Family

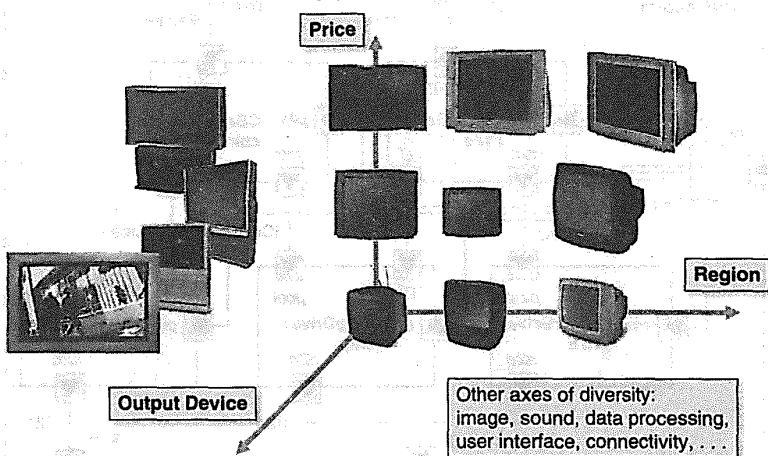


Figure 1-5.
Philips TVs as a
product family,
varying along
three dimensions.
(Figure courtesy
Royal Phillips
N.V.)

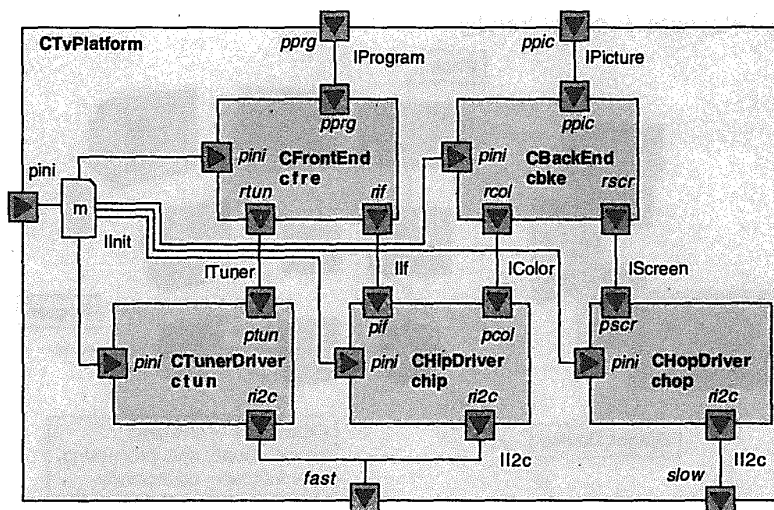
Philips addressed the problem by formulating a product family, as shown in Figure 1-5, in which its many different types of TV sets varied along three dimensions—price, output device, and geographical region—while they shared their basic purpose and much of their functionality. The same approach was applied to other Philips products including VCRs, DVD players, and audio equipment.

In order to support its product families adequately, Philips had to be able to address two key issues: *commonality* and *variability* across products. The key observation that the Philips engineers made, and one that is of particular relevance to this book, was that the product family notion extended to the growing amounts of *software* embedded in the different devices. To exploit the commonality and manage the variability across the different embedded software product families, Philips developed an architectural methodology, called Koala. A more extensive treatment of Koala is provided in Chapters 6 and 15; here we only highlight its key features.

Koala models and implements a software system as a collection of interacting components. Each component exports a set of services via a set of *provides* interfaces. Additionally, each component explicitly defines its dependencies on its environment (either the hardware or other software components) via a set of *requires* interfaces. For illustration, a software architecture for a Philips TV platform modeled in Koala's graphical notation is shown in Figure 1-6.

This approach allows an engineer to construct and analyze an architecture with relative ease: Each component is essentially akin to a Lego block with well-defined “pins” for composing with other components; in order for such compositions to be legal, the *provides* and *requires* interfaces of the respective components must match according to a set of rigorously defined rules. Moreover, a given assembly of components in Koala can be treated as a *compound* component, which can then be used as a single unit. For example, the architecture shown in Figure 1-6 is a compound component with three *provides* and two

Figure 1-6.
An example
software
architecture for a
TV set. The
architecture
consists of five
interacting
components with
various incoming
and outgoing
interfaces.
(Figure courtesy
Royal Phillips
N.V.)



requires interfaces. This allows components of arbitrary complexity to become reusable assets across Koala architectures (that is, across Philips products).

Our previous examples of the use of software architecture, the Web and pipe-and-filter applications, hint that a focus on architecture is a focus on reuse: reuse of ideas, knowledge, patterns, and well-worn experience. In turn, product family architectures facilitate a higher-order level of reuse: reusing structure, behaviors, implementations, and so on, across many related products.

In addition to its ability to exploit commonalities within a product family via reusable assets, Koala also provides explicit support for managing variability across products. Three separate mechanisms are used to this end.

1. *Diversity interfaces* are a mechanism for parameterizing a component. Diversity interfaces allow a component to import configuration-specific properties from Koala's specialized interface implementation elements, which are external to the component and are contained within the architecture encompassing the component.
2. *Switches* are connecting elements that allow a single component to interact with one of a set of components, depending on the value of a given run-time parameter.
3. *Optional interfaces* allow a component of a given type either to provide or to require additional functionality that may be specialized for certain, but not all, products within a family.

The combination of Koala's support for sharing reusable assets of arbitrary complexity and managing diversity across products in a family has made it possible for Philips to explore combining components from different families in novel and interesting ways. Some of those are depicted in Figure 1-7. The resulting *product populations* are a further extension of the technical challenges inherent in product families. For example, the presence of

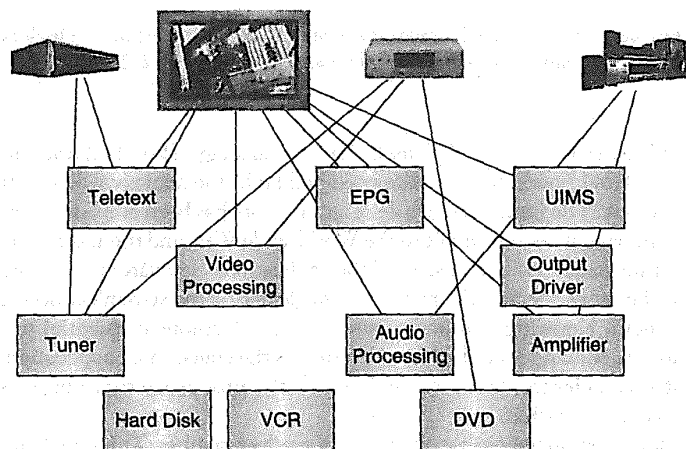
Composition

Figure 1-7.
An ever-expanding range of possibilities opens by combining existing consumer electronics products. This provides an even broader and longer-term motivation for product families. (Figure courtesy Royal Philips N.V.)

variation induces requirements for extensive configuration management of the individual components and of entire architectures.

Koala has directly helped Philips to conquer these challenges and turn them into a competitive advantage. We should note that while architecture has played a central role in this process, product families and populations also require broader changes to processes and business organization practices. The standard approach to software development as found in most businesses does not effectively support product families. One common reason is that a development team often has little incentive to expend its already scarce resources and produce more widely (re)usable assets that will benefit other teams and projects in the future. Therefore, introduction of a robust strategy to support product families also demands changes in how development organizations are structured and operate internally, and how they interact with other parts of a company's business, including marketing, hardware engineering, and finance.

Apart from the obvious cost savings from the reuse of components and the ability to quickly craft yet another new television configuration, the Philips example reveals another critical benefit of the software architecture abstraction: Koala is the concrete manifestation of the company's corporate experience, knowledge, and competitive advantage. Without a specific way of representing such knowledge, companies are left relying on the memories of their employees and the verbal conveyance of past experience to retain and pass along "our company's way" of solving problems and creating new products. With increasing trends of employee mobility from company to company, reliance on a social structure to maintain such understanding is an increasingly shaky policy. Software architecture offers not only a solution to a socially induced problem, but, with its substantial technical support, the ability to achieve much higher levels of productivity than have heretofore been seen. This is a critical observation and further explication of it will be provided throughout the remainder of this book.

1.5 END MATTER

**Vitruvius
on the
Character of
Architecture**

Architecture depends on fitness (*ordinatio*) and arrangement (*dispositio*), . . . ; it also depends on proportion, uniformity, consistency, and economy, . . . Vitruvius I, 2, 1

This chapter began with an extended discussion of an analogy of how buildings are designed and constructed and how software is designed and built; the analogy will appear again in following chapters. The fundamental role of architecture has been emphasized throughout the chapter, showing its essential role in the World Wide Web, and then, at the other end of the size spectrum, in quickly composed desktop applications. Software architecture has been presented as the set of *principal design decisions* made during the system's conceptualization and development. Knowing and understanding these decisions is essential to successful system evolution. Making such decisions determines the course of an application: whether it will be effective, elegant, and successful or a costly, error-prone thorn in the side of all who come in contact with it.

The chapter concluded with a discussion of product families. This concluding focus is particularly appropriate as it echoes closely a lesson from the world of buildings, namely the critical role of architecture and architects in carrying forward lessons, experiences, and reusable elements to new projects.

In 1979, architect Christopher Alexander's book, *The Timeless Way of Building*, showed how patterns of solutions to common architectural needs have developed over the years, and how combinations of patterns can be made to yield structures that address complex needs and constraints. Software architecture not only supplies the intellectual means for applying such reuse of knowledge in the software domain, enabling engineers to efficiently address needs for new systems, but also provides the framework for reusing substantial functional components in the production of new members of a product family.

The vision of architecture-centric software engineering is very compelling. It prefigures a world where complex software systems are engineered, rather than crafted, drawing from extensive experience in past projects to form the basis for new ones. System designs are modeled in a variety of notations, each optimized for depicting particular aspects of the architecture. Powerful tools automate the process of maintaining consistency between these models. Each project stakeholder has a panoply of visualizations available for looking at the architecture in ways that are most natural or convenient. Capable analysis tools provide deep insights into the nature of the designed system long before implementation activities begin—stakeholders are able to assess the qualities of a software system from its design before costly mistakes are introduced. The design serves as the basis for system deployment and evolution as well, informing future engineers as to exactly how to evolve the system in ways that are consistent with its original design principles and goals. The architecture (and the qualities induced by it) accompanies the system through its lifetime to its eventual retirement. This vision is not fully a reality today. However, as this book will show, today's foundations, theory, and practice of software architecture can achieve vastly superior results to those of traditional development practices, while also providing a rich basis for achieving the stated vision.

The coming chapters will supply the missing details in our discussion. To apply the techniques of software architecture one must have adequate conceptual foundations, notations, analysis techniques, tools, and processes, all of which will be discussed. We will begin this examination in the next chapter by considering how a software architecture-centric approach to development radically reshapes the other tasks and processes of software engineering. The emphasis will be on recognizing that principal design decisions are made throughout a system's life cycle, and that by focusing on such decisions, capturing the knowledge they represent, the development process is improved. Chapter 3 then sets the concepts of software architecture in precise terms, while subsequent chapters show how to design, develop, deploy, and adapt systems based on these principles.

1.6 REVIEW QUESTIONS

1. What are the principal insights from the discipline of building architecture that are applicable to the construction of software systems?
2. Recognizing the limitations of an analogy—where it does not apply—can be as instructive as considering the situations where it does apply. What's wrong with the building architecture analogy? In what ways is constructing software fundamentally different from building a physical structure? Do a Web search for "software construction analogy is broken." Do you agree or disagree with the opinions posted online?
3. Philips's use of software architecture for supporting product lines is targeted at consumer electronics, such as televisions. What other industries or markets could benefit from the commonalities and efficiencies of a software product-line approach?

1.7 EXERCISES

1. Architects use hand-drawn sketches, prose, and blueprints to describe buildings. What do these correspond to in software development?
2. When a software developer begins a new development task by directly starting to program, what kind of development activity would that correspond to in building?
3. Look up several definitions of "software design." Do any of these definitions correspond to what architects (or industrial designers) term "design"?
4. What corresponds in software development to a building architect's concern for aesthetics?
5. Interview an architect and find what his or her key vocabulary items are. From your knowledge of software engineering, what analogs do those terms have in software?
6. Is X a better analogy for the construction of software? Why or why not? Let X = (law, medicine, automotive engineering, oil painting). For example, in law, consider the process by which laws are made and enforced. For oil painting, consider the commissioned works that artists such as Michelangelo produced. What process did he follow? For medicine, consider whether the "product" of medicine is tangible or not, and what that implies for medical knowledge.
7. Perform a Web search for "software architecture." Are the top hits deserving of the billing? Are the terms used on those Web sites consistent with the terms as used in this textbook?
8. Write a pipe-and-filter application that prints a sorted list of every unique word in a file. That is, each unique word should appear only once on the output.

1.8 FURTHER READING

The works of Vitruvius have been translated into English many times and should be readily available in a good

library. Free, online translations are also available; see, for example, Bill Thayer's Web site (Pollio).

Many excellent books on architecture have been written, of course, but few provide substantive insights for software developers. A significant exception is Stewart Brand's *How Buildings Learn* (Brand, 1994). Brand chronicles how a wide range of kinds of buildings have been changed over the many years since their initial construction. A discussion of some of the principles from this book and how they relate to software architecture is found in Chapter 14 of this text. Another architecture text that has had substantial impact on software development is Alexander's *The Timeless Way of Building* (Alexander, 1979). This work has influenced thinking on patterns for object-oriented programming, and more generally, software architecture styles.

An interesting chronicle of the construction of a New York City skyscraper can be found in (Sabbagh, 1989).

The processes the designers and builders follow, and the problems they encounter, are eerily similar to those in large-scale software development.

A detailed discussion of the architecture of the World Wide Web can be found in (Fielding and Taylor 2002). Key points from this work are discussed in detail in Chapter 11. The standards that govern the operation of the Web are available from the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) Web sites: www.ietf.org and www.w3.org, respectively. Tim Berners-Lee's description of the Web from 1994 can be found at (Berners-Lee, et al. 1994).

Further information on Koala and how it has been used to support development of product families is available at (van Ommerring et al. 2000) and (van Ommerring 2002).

Architectures in Context: The Reorientation of Software Engineering

The preceding chapter introduced the concept of software architecture and illustrated its power in establishing the contemporary World Wide Web. It also showed how software architecture provides a technical foundation for the exploitation of the notion of product families. This chapter shows how software architecture relates to the concepts of software engineering that traditionally are the most prominent.

What emerges from this consideration is a reorientation of those concepts away from their typical understanding, for the power of architecture demands a primacy of place. As a result the very character of key software engineering activities, such as requirements analysis and programming, are altered and the technical approaches taken during various software engineering activities are changed.

The focus of this chapter is showing the role of architecture in the whole of the software engineering enterprise, so it is somewhat cursory—most of the major topics of software engineering are discussed, but each individual topic is only allotted a few pages. Subsequent chapters will take up the major points in turn, treating them in substantial detail. The reader will be able to understand the role of software architecture in the larger development context, and could apply the ideas in broad terms, but without the specific techniques and tools covered in subsequent chapters effective application of the ideas will be elusive.

**Outline of
Chapter 2**

2 Architectures in Context: The Reorientation of Software Engineering	2.6 Evolution and Maintenance
2.1 Fundamental Understandings	2.7 Processes
2.2 Requirements	2.7.1 The Turbine Visualization
2.3 Design	2.7.2 Example Process Depictions
2.3.1 Design techniques	2.8 End Matter
2.4 Implementation	2.9 Review Questions
2.4.1 Implementation strategies	2.10 Exercises
2.5 Analysis and Testing	2.11 Further Reading

2.1 FUNDAMENTAL UNDERSTANDINGS

There are three fundamental understandings of architecture, the recognition of which helps situate architecture with respect to the rest of software engineering:

1. Every application has an architecture.
2. Every application has at least one architect.
3. Architecture is not a phase of development.

By architecture we mean the set of principal design decisions made about a system; it is a characterization of the essence and essentials of the application. Referring back to the analogy to buildings discussed at the beginning of Chapter 1, it is evident that every building has an architecture. That is not to say that every building has an honorable, or elegant, or effective architecture: Sadly, many buildings evince architectures that ill-serve their occupants and shame their designers. But those buildings *have* architectures, just as much as the beautiful, elegant buildings that are a delight to see and to work within. So it is with software. An application's fundamental structure can be characterized, and the principal design decisions made during its development can be laid out. For example, as described in the previous chapter, the architecture of the World Wide Web is characterized by the set of decisions referred to as the REST architectural style. The Unix shell script we saw in the previous chapter has an architecture characterized by the pipe-and-filter style. All applications do have architectures because they all result from key design decisions.

The contribution of this observation is that it immediately raises questions: Where did the architecture of an application come from? How can that architecture be characterized? What are its properties? Is it a "good architecture" or a "bad architecture?" Can its shortcomings be easily remedied?

The second understanding follows naturally from the first: Every application has an architect, though perhaps not known by that title or recognized for what is done.³ The architect is the person or, in most cases, group who makes the principal decisions about the application, who establishes and (it is hoped) maintains the foundational design. As with

³Conversely, just because a company bestows a title such as "chief software architect" on someone does not necessarily mean that the individual possesses any particular credentials or ability to responsibly create good or effective software architectures, or indeed to know anything about software technology.

the first understanding, this observation also immediately raises some questions, including: Were the architects always aware when they had made a fundamental design decision? Can they articulate those foundational design decisions to others? Can they maintain the conceptual integrity of the design over time? Were alternatives considered at the various decision points?

The third understanding is the most profound, and which provides the focus for the majority of this chapter. Architecture refers to the conceptual essence of an application, the principal decisions regarding its design, the key abstractions that characterize the application. If it is agreed that every application has an architecture, then the question arises, Where did it come from? How does it change?

In a simplistic, traditional, and *inaccurate* understanding, the architecture is a specific product of a particular phase in the development process that succeeds identification of requirements and that precedes detailed design. In elaborate software engineering waterfall processes, or in the spiral model, such phases are often explicitly labeled, using such terms as “preliminary design,” “high-level design,” “product design,” or “software product design.” Thinking of architecture as the product of such a phase—especially an early phase—confines architecture to consist of only a few design decisions, a subset of those that fully characterize the application, and, unfortunately in many cases, to the decisions most likely to be contravened by subsequent decisions. While the creation and maintenance of the architecture may begin or have special prominence in a particular phase, these activities pervade the development process.

To show how software architecture relates to the broader traditional picture of software engineering, we discuss architecture in the context of traditional phases and activities of software engineering. For convenience and familiarity, we organize the following discussion around the notional waterfall process. Our discussion is predicated on a recognition that a system’s architecture should result from a conscious, deliberate activity by an individual or group recognized as having responsibility for the architecture. Those unfortunate situations where the architecture is not explicit and not the result of conscious choice or effort simply reflect poor engineering practice.

2.2 REQUIREMENTS

Consideration of architecture rightly and properly begins at the outset of any software development activity. Notions of structure, design, and solution are quite appropriate during the requirements analysis activity. To understand why, it is appropriate to begin with a short exploration of the traditional software engineering approach. We contrast that view with the practice in actual software development, the practice in (building) architecture, and then consider the role of design and decision making in the initial phase of application conception.

In the waterfall model, indeed in most process models, the initial phase of activities is focused on developing the requirements for an application—the statement of what the application is supposed to do. The traditional academic view of requirements analysis and specification is that the activity, and the resulting requirements document, should remain unsullied by any consideration for a design that might be used to satisfy the identified requirements. Indeed, some researchers in the requirements community are rather strident

on this point: “The central part of this paper sketches an approach to problem analysis and structuring that aims to avoid the magnetic attraction of solution-orientation” (Jackson 2000); similar quotes can be found throughout the literature. (More recent work in requirements engineering has moved away from such a viewpoint, as will be discussed at the end of the chapter.)

The focus on isolating requirements and delaying any thought of solution goes back to antiquity. Pappus, the Greek mathematician from the fourth century wrote, “In analysis we start from what is required, we take it for granted, and we draw consequences from it, till we reach a point that we can use as [a] starting point in synthesis.” Synthesis is here the design, or solution, activity. More recently, the twentieth-century American mathematician George Polya wrote an influential treatise on “How to Solve It” (Polya 1957). The essence of his approach is as follows.

- First, understand the problem.
- Second, find the connection between the data and the unknown. You should obtain eventually a plan of the solution.
- Third, carry out your plan.
- Fourth, examine the solution obtained.

In both these approaches, a full understanding of the requirements precedes any work toward solution. Most software engineering writers have taken this essential stance, as we have seen in the quote above, and argued that any thought about how to solve a problem must follow a full exploration and understanding of the requirements.

One of the stories used to illustrate the wisdom of this approach concerns the development of washing machines. If in designing washing machines we simply automated the manual solution of ages past we would have machines that banged clothes on rocks down by the riverside. In contrast, focusing on the requirements (namely, cleaning clothes) independent of any “magnetic attraction of solution-orientation” allows novel, creative solutions to be obtained: rotating drums with agitators.

This ideal view of isolating requirements first, then designing afterward, is in substantial contrast to the typical practice of software engineering. While it is not something that developers write academic papers about, the practice indicates that, apart from government-contracted development and some specialized applications, substantive requirements documents are seldom produced in advance of development. Requirements analysis, which ostensibly produces requirements documents, is often done in a quick, superficial manner, if at all. Explanations for such deviation of actual practice from putative “best practices” include schedule and budget pressures, inferior processes, denigration of the responsible engineers’ qualifications and training, and so on.

We believe the real reasons are different, and that these differences begin to indicate the role of architecture and solution considerations at the very outset of a development process. The differences have to do with human limitations in abstract reasoning, economics, and perhaps most importantly, “evocation.”

Consider once again the analogy to buildings. When we decide we need a new house or apartment, or a modification to our current dwelling, we do not begin a process of reasoning about our needs independently of how they may be satisfied. We think, and talk, in terms of how many rooms, what style of windows, whether we want a gas or electric stove—not in

terms of “means for providing shelter from inclement weather, means for providing privacy, means for supplying adequate illumination, means for preparing hot food” and so on. We have extensive experience with housing, and that experience allows us to reason quickly and articulately about our desires, and to do efficient approximate analyses of cost and schedule for meeting the needs. Seeing other houses or buildings inspires our imagination and sparks creative thought as to what “might be.” Our needs start to encompass particular styles of houses that we find charming or cutting edge. Thus our understanding of the architecture of our current dwelling and the architecture of the other people’s dwellings enables us to envision our “needs” in a way that is likely to lead to their satisfaction (or revision downward to reality!)

So it is with software. Specifying requirements independently of any concern for how those requirements might be met leads to difficulty in even articulating the requirements. Aside from limited domains, such as numerical analysis, it is exceptionally difficult to reason in purely abstract terms. Without reference to existing architectures it becomes difficult to assess practicality, schedules, or cost. Seeing what can be done in other systems, on the other hand, what kind of user interfaces are available, what new hardware is capable of, what kind of services can be provided, evokes “requirements” that reflect our imagination and yet are grounded in reasonable understandings of possibility. Current practice reflects this approach, showing the practicality of reasoning about structure and solution hand-in-hand with requirements.

This observed experience with software practice is consistent with that found in other engineering disciplines. Henry Petroski, the popular chronicler of the modern history of engineering, has observed that failure is the driver of engineering and the basis for innovation. That is, innovation and new products come from observation of existing solutions and their limitations. Reference to those prior designs is intrinsically part of work toward new solutions and products.

The nearly ubiquitous zipper followed a typical engineering development path. Rather than starting from a design-free functional specification—something like “means for joining edges of a jacket”—the zipper appeared in the early 1900s as an incremental successor to a long line of patented inventions. The earliest of these go back to the problem of buttoning high-top boots. From the C-curity hook-and-eye fastener to the Plako to the hidden hook and then to the still-common nested, cup-shaped fasteners we refer to as zippers, the history is one of repeated invention, failure, and new innovation. In addition to the technical challenges of getting a fastener to function reliably and be aesthetically acceptable, the developers of the zipper also faced the equally daunting challenge of developing a demand for the product.

Henry Petroski tells the story in detail in *The Evolution of Useful Things* (Petroski 1992). He concludes his chapter on the zipper with the following insight, “... like the form of many a now familiar artifact, that of what has come to be known as the zipper certainly did not follow directly from function. The form clearly followed from the correction of failure after failure.”

The Development of Zippers

Similarly, if we consider the behavior of corporations, or especially of venture capitalists, we see a focus from the outset on structures and designs, rather than requirements. That is, one does not successfully approach a venture capitalist and simply enumerate a great list of requirements for some product. Requirements do not create value; products do. Successful new ventures are begun on the basis of a potential solution—possibly even without

a conception of what requirement it might be meeting. Marketing organizations within companies have as their charter the responsibility to match existing products to needs, to create needs, or, especially, to identify how existing products need to be modified in order to address emerging customer needs. In all this, solution and structure are equal partners with requirements in a conversation about needs. The core observations from this reflection are:

- Existing designs and architectures provide the vocabulary for talking about what might be.
- Our understanding of what works now, and how it works, affects our wants and perceived needs, typically in very solution-focused terms.
- The insights from our experiences with existing systems helps us imagine what might work and enables us to assess, at an early stage, how long we must be willing to wait for it, and how much we will need to pay for it.

The simple conclusion then, is that analysis of requirements and consideration of design—concerning oneself with the decisions of architecture—are naturally and properly pursued cooperatively and contemporaneously.

The starting point for a new development activity thus includes knowledge of what exists now, how the extant systems fail or fail to provide all that is desired, and knowledge of what about those systems can or cannot be changed. In many ways, therefore, the current architectures drive the requirements. Indeed, requirements can be thought of as the articulation of improvements needed to existing architectures—desired changes to the principal design decisions of the current applications. Architectures provide a frame of reference, a vocabulary, a basis for describing properties, and a basis for effective analysis. New architectures can be created based upon experience with and improvement to pre-existing architectures.

The overwrought objection to this approach is that it will limit innovation and guide development down unfruitful paths. After all, that is the point of the washing machine story mentioned earlier. The problem is, while the anecdote is amusing, it does not reflect how modern washing machines were developed. Engineers at Maytag did not begin by stating abstract requirements for the “removal of particulate matter greater than n microns in size from fabrics composed of cotton fibers, to an effective level of only k residual particles per gram of fabric after t seconds of processing.” Rather, history shows an incremental progression of machines that largely draw from automation of formerly human actions, but with innovative progressions.

Developers, nonetheless, do have to be concerned about having their vision limited by current designs. An analogy used by professors David Garlan and Mary Shaw makes the point well: To ascend to the top of a house a ladder may be used. To ascend to the top of a small building, a fire truck ladder may be used. But to ascend to the top of a skyscraper a ladder of any variety will be insufficient; an elevator (which bears no physical relationship to a ladder) is effective. To ascend to the moon requires yet another totally different technology. In this analogy, the concept of ascension is constant across the different situations, but the specifics of the situations end up implying the need for distinctly different solution architectures.

The point is this: Predecessors, that is, existing architectures, provide the surest base for the vast majority of new developments; in such situations requirements should be stated

using the terminology and structural concepts of existing systems. In situations where an adequate (“physical”) predecessor is *unknown*, conceptual predecessors or analogies (for example, “things used before to ascend”) may provide a terminological basis for describing the new needs, but caution must be exercised in drawing any architectural notions from such predecessors. Conceptual predecessors may provide a vision but frequently offer little help beyond that.⁴

Perhaps most dangerous is so-called greenfield development: one for which there is no immediate architectural predecessor. In such cases the temptation is to believe that new conceptual ground is being broken and hence no effort need be or should be undertaken to examine existing systems and architectures for framing the development. The risk is a directionless development; better is the strategy to first extensively search for existing architectures that might provide the efficient basis for framing the new requirements and solution. Greenfields are often minefields in disguise.

Lastly, the admonition to always consider architectural predecessors and to base development on improvements to existing systems is not absolute. Not all architectures are worthy of further work and may be incapable of serving as the basis for additional development. As further chapters will illustrate, some architectural styles are much more effective at supporting change and enhancement than others; some architectures deserve to be abandoned.

This section has called for a new understanding and approach toward requirements engineering which provides substantial prominence to the role of architectures and solution considerations. Architectures provide the language for discussion of needs—not only a glossary of terms, but a language of structures, mechanisms, and possibilities. With this architectural underpinning preliminary analyses can proceed. Requirements documents still serve an important role in the engineering process, for instance as the basis for contracts and setting out the precise objectives for new work, but these documents should be strongly informed by architectural considerations. Chapter 15 will elaborate an advanced application of these ideas known as reference requirements, under the banner of “domain specific software architectures.” Later in this chapter, we briefly return to the relationship between requirements and development, when in the section on processes we consider the Twin Peaks model and agile development methods.

2.3 DESIGN

Designing is, by definition, the activity that creates a system’s software architecture—its set of principal *design* decisions. As the preceding sections have indicated, these decisions are not confined, for instance, to high-level notions of a system’s structure. Rather the decisions span issues that arise throughout the development process. Nonetheless, there is a time during development when more attention is paid to making many of those principal decisions than at other times—an architectural design “phase.” Our points for this section

⁴Nonetheless, the considerable inspiration that can come from considering an analogy from a different domain of knowledge should not be underestimated. An example *par excellence* is the invention of hypertext: Vannevar Bush in his article, “As We May Think” (Bush 1996) conceived of the notion of hypertext through explicit analogy to how the human brain operates associatively. His conception for a device for making and recalling information associatively, however, had nothing to do with biology or cells, but rather with extrapolations from mechanical devices of the 1940s.

are simply as follows:

- The traditional design phase is not *exclusively* “the place or the time” when a system’s architecture is developed—for that happens over the course of development—but it is a time when particular emphasis is placed on architectural concerns.
- Since principal design decisions are made throughout development, designing must be seen as an aspect of many other development activities.
- Architectural decisions are of many different kinds, requiring a rich repertoire of design techniques.

In traditional formulations of software engineering, the activity of design is confined to the “design phase.” In the notional waterfall model, the abstract requirements produced by the requirements phase are examined, a design process is followed, and, at completion of the phase, a design is produced and handed to programmers for implementation. Most often those decisions concern a system’s structure, including identification of its primary components and how they are connected. (Hence such a design denotes a particular set of design decisions that *partially* comprise the architecture.) To the extent that, during the design phase, some portion of the requirements are found to be infeasible, those requirements issues are referred back to the requirements phase for reconsideration. (And, in the extreme purist view, without passing back any solution information!) If any portions of the design are found to be infeasible or undesirable during the implementation phase, those issues are referred back to the design phase for reconsideration.

With an architecture-centric approach to development, these artificial and counterproductive phase boundaries are diminished or eliminated. As the previous section discussed, the analysis of requirements is properly bound up with notions of architecture and design, with these notions providing the context and vocabulary for describing the new capabilities desired. The activities of analysis, design, and implementation proceed, but in an enriched and more integrated fashion. The principal decisions governing an application—its architecture—are informed from many sources and typically made throughout the process of development.

Last, a rich repertoire of design techniques is needed to assist the architect in making the wide range of design decisions comprising an architecture. The architect must deal, for example, with:

- Stakeholder issues, such as choices concerning the use of proprietary, commercial off-the-shelf or open-source components, with their attendant and varying licensing obligations.
- Over arching style and structure.
- Types of connectors for composing subelements.
- Package and primary class structure (that is, low-level style and structure when working in an object-oriented development context).
- Distributed and decentralized system concerns.
- Deployment issues.
- Security and other nonfunctional properties.
- Postimplementation issues (means for supporting upgrade and adaptation).

Clearly, a simple one-size-fits-all design strategy will not suffice. The following subsection considers the topic of design techniques in a little more detail. In many ways, however, everything in the rest of this book is an exploration of the many facets of software design; this section provides only the briefest introduction.

2.3.1 Design Techniques

A variety of strategies exist for helping the designer develop an architecture. Chapter 4 will discuss several of them in substantial detail, including the use of basic conceptual tools, architectural styles and patterns, and strategies for dealing with unprecedented systems. For the purposes of this chapter, namely relating an architecture-centric approach with classical software engineering, we only consider two strategies for use in development of a solution architecture: object-oriented design and the use of domain-specific architectures, just to illustrate the spectrum of approaches, issues, and concerns.

Object-oriented design is perhaps the most common design approach taught and close consideration of it helps reveal why a wide repertoire of techniques is necessary. A similar analysis could be developed for other traditional approaches to design, such as functional decomposition.

Object-Oriented Design

Object-oriented design (OOD) is usually taught in the context of object-oriented programming, that is, the reduction of an algorithm to machine-executable form. The essence of OOD is the identification of so-called objects, which are encapsulations of state with functions for accessing and manipulating that state. Numerous variations are found in different object-oriented programming languages regarding the way objects are specified, related to one another, created, destroyed, and so on. Given the prevalence of the technique it will not be further summarized here; numerous excellent references are widely available (Larman 2002; Schach 2007).

While object-oriented design can be used as a strategy when developing an architecture, it is not a complete approach and is not effective in all situations. This is not to say that it is an ineffective design technique. On the contrary, OOD is exceptionally effective in a wide variety of development situations! It is just important to realize its limitations, so that other additional or alternative techniques can be brought to bear on those portions of the task.

First, clearly, OOD is not a complete design approach, for it fails to address myriad stakeholder concerns. OOD says nothing about deployment issues, security and trust, or use of commercial components, for instance. Similarly, OOD, as a practice, has no intrinsic means for carrying forward domain knowledge and solutions from previous architectures to new products. While code reuse techniques may be employed or higher-level representations such as Unified Modeling Language (UML) used, no facility is available for explicitly indicating points of variation or other aspects of program families.

Second, OOD is not effective in all situations. For instance, some of its limitations are as follows:

- OOD views all applications as consisting, solely, of one software species (namely, “object”), regardless of purpose. Forcing all concepts and entities into a single mold can obfuscate important differences. The REST style, for instance, maintains distinct

notions of “user agent,” “origin server,” and “cache.” Apart from using programming language types to try to characterize these distinct elements, OOD does not offer any helpful modeling mechanism.

- OOD provides only one kind of encapsulation (the object), one notion of interface, one type of explicit connector (procedure call), no real notion of structure (objects are constantly created and destroyed), and no notion of required interfaces. This implies that all the richness of a given solution approach has to be mapped down or transformed to this level of single choice. Some architectures—such as pipe-and-filter—may not be at all effective after such a transformation.
- OOD is so closely related to programming language concerns and choices that it tends to have the forest obscured by the trees. For instance, the vagaries of type inheritance may obscure identification of the principal objects critical to the design. Similarly, it is so bound with programming language issues that the language may start dictating what the important decisions are.
- The typical OOD approach assumes a shared address space and adequate support for heap-and-stack management. OOD typically assumes a single thread of control; support for multiple threads is accommodated in languages largely as an afterthought. Concern for concurrent, distributed, or decentralized architectures is largely outside the OOD purview. Support for distributed objects, such as that provided by CORBA-style middleware, attempts to hide the presence of network boundaries. Architectures that must deal with the realities of networks and their unavoidable characteristics of introducing failures, latency, and authority domains will not be directly supportable with an OOD approach.

One positive step in the object-oriented design world has been the creation of the UML notation. This has helped lift the discussion of object-oriented designs above the programming language level. In 2003, the UML notation was enhanced with some simple notions of software architecture, further improving its contribution. UML is covered in Chapters 6 and 16.

Also helping to lift object-oriented design closer to a richer notion of architecture is the extensive work in patterns, as exemplified in the work of computer scientists Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gamma et al. 1995). Patterns are explicitly designed to enable the carry-forward of knowledge and experience from previous work into new applications. As such, patterns represent an excellent, “in the small,” OO-particular application of an important concept from software architecture.

Extending the idea of patterns to a broad, application-encompassing level yields a design approach known as domain-specific software architectures. This approach exemplifies how architecture-centric design may be supported, and how significant technical and business benefits can consequently be achieved.

Domain-Specific Software Architectures (DSSAs)

The approach known as domain specific software architectures, or DSSAs, is appropriate when prior experience and prior architectures are able to strongly influence new projects. The key notion is that if a developer or company has worked within a particular application domain for many years, it is likely that a best approach or best general solution for applications within that domain will have been identified. Future applications in that domain can

leverage this knowledge, with those applications having their fundamental architecture determined by the architectures of the past generations of applications. Indeed, the new architectures will likely be variations on the previous applications. The technical essence of the domain-specific software engineering approach is to capture and characterize the best solutions and best practices from past projects within a domain in such a way that production of new applications can focus more or less exclusively on the points of novel variation. For those points where commonality with past systems is present, reuse of those corresponding parts of the architecture, and indeed reuse of corresponding parts of the implementation, is done. The DSSA approach to system development is thus consistent with the development of product lines as introduced in Chapter 1, and can be the technical centerpiece of a product-line approach.

For instance, if a company is in the business of producing television sets for the world market, such as the example of Philips discussed in Chapter 1, a backbone software architecture can be identified that can be used in televisions destined for sale in North America, Europe, and the Far East. Different tuners for the different markets can be written, and if the backbone architecture has identified variation points and interfaces for the tuner component, the various market-specific tuners can be slotted in, enabling customization in a very cost-effective manner.

To effectively follow the DSSA approach, good technical support is required: The architecture of the previous generation of applications must be captured and refined for reuse; the points of allowable variation must be identified and isolated, the interfaces between the points of variation and the core architecture must be made explicit, the dependencies between multiple points of variation must be identified, and so on. A particularly important element of the DSSA approach is creation and use of a standard way of describing products and systems within the domain of interest. Use of regularized terminology enables an engineer to determine the degree to which a new product concept fits within the company's existing DSSA. These issues and more are explained in Chapter 15, where the DSSA concept and product lines are explored in depth; suffice it to say here that the DSSA approach to design enables the many advantages of an architecture-centric development approach to be realized.

2.4 IMPLEMENTATION

The task of the implementation activity is to create machine-executable source code that is faithful to the architecture and that fully develops all outstanding details of the application. This is true whether speaking from the perspective of classical software engineering or from the software architecture-centric view of software engineering set forth in this book. Architecture-centric development somewhat changes our understanding of the implementation activity and its responsibilities, however. It also emphasizes some particular approaches to implementation.

First, the implementation activity may add to or modify the architecture—if key design decisions are made while working with the source code, they are part of the architecture as much as any principal decision made much earlier in the process. Second, there is no presumption that architecture is completed before implementation begins. Indeed there may be substantial iteration and revision of the architecture as the code is developed,

depending on the development process followed. What the architecture approach emphasizes is keeping all the recorded decisions constituting the architecture consistent with the code as the application is fully developed. That is, the implementation activity cannot turn its back upon the preceding decisions and proceed to modify, for example, the application's structure without making corresponding changes to the recorded architecture.

With regard to implementation strategies, architecture-based development recognizes the enormous cost and quality benefits that accrue from generative and reuse-based implementation strategies. In those cases where full automatic generation is not possible—by far the dominant case—a set of supportive techniques are emphasized. These include application frameworks and other types of code reuse, which provide significant parts of the implementation prebuilt.

The watchword for the creation of the implementation is its being faithful to, or consistent with, the architecture. The simplest understanding of a faithful implementation is that all of the structural elements found in the architecture are implemented in the source code, and that all of the source code corresponds to various parts of the explicit architecture. It is fair for the source code to contain more details, such as low-level algorithms for implementing a higher-level concept described in the architecture, but the source code:

- Must not utilize major new computational elements that have no corresponding elements in the architecture.
- Must not contain new connections between elements of the architecture (for example, such as might be motivated by efficiency concerns) that are not found in the architecture.

This initial conception of the relationship between the architecture and the source code is not fully adequate, however, for it does not accommodate some of the legitimate characteristics of reuse-oriented engineering. For instance, an architecture may require the presence of a component to perform some mathematical functions—a sine and cosine routine perhaps. The most cost-effective and quality-focused approach to providing an implementation for that component may be to acquire and use a general math library, one providing not only sine and cosine functions, but tangent, inverse trig functions, and so on. Such a component may have interfaces in addition to those required by the architecture, and will certainly contain subfunctions that do not correspond to anything identified in the architecture.

Further issues may come into this notion of faithfulness between the implementation and the architecture as it existed before the implementation began. Examination of a candidate mathematics library may reveal that it provides 98 percent of the functionality required by the architecture, for a very low price, and at a very high quality level. Analysis of the consequences of the missing 2 percent of the functionality may lead the designers to conclude that the missing functionality will be acceptable under most usage situations, and for the cost savings realized, (1) choose to use the library, and (2) revise the specifications of the system, including the architecture, in accordance with the reduced functionality. The critical point is that the architecture is always kept in a consistent state with the implementation.

2.4.1 Implementation Strategies

A variety of techniques are available to assist in the faithful development of the implementation from the design, including generation technologies, frameworks and middleware, and reuse. Choice of the techniques is largely determined by availability. Generative techniques, when available, are often the best: The implementation is generated automatically and is of very high quality. The reuse-based techniques come next: Implementation time is significantly less than programming the entire implementation from scratch and quality is similarly better. Least desirable is writing all the code manually: The project cost goes up significantly due to extended development and quality-assurance time. These various approaches are discussed below.

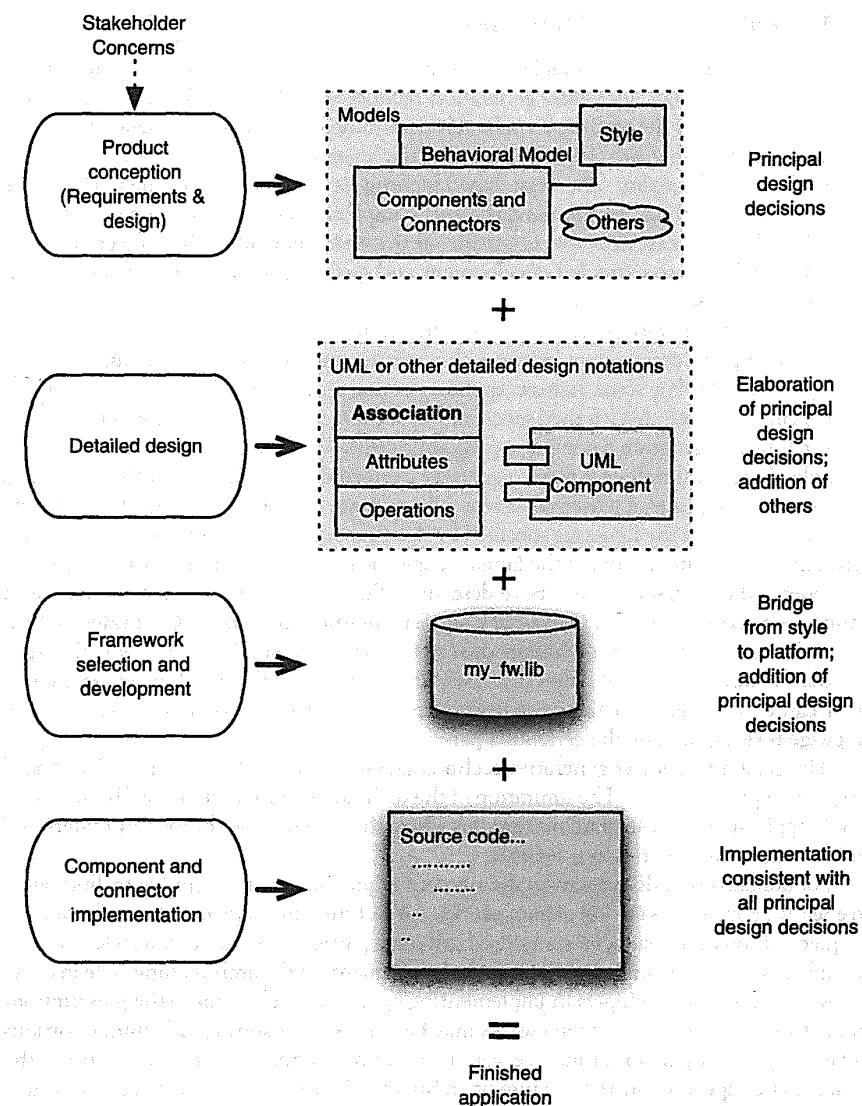
The use of generation technologies is the easiest and most straightforward way of ensuring complete consistency between the architecture and the implementation. The concept is simple: For some narrow application domains, sufficient intelligence can be encoded in a tool such that a formal specification of the architecture can be input to the tool, and the tool will generate a complete program to perform the functions as specified. One example of such technology is parser generators. Given the specification of the syntax of a programming language, a parser generator will generate a program capable of recognizing programs written in the language specified by the grammar. Choice of the parser generator determines the required form of the language specification and determines the structure of the generated parser, such as recursive descent or SLR(1) (for a list of such tools, consult online resources such as *The Catalog of Compiler Construction Tools 2006*) or (Heng). Bear in mind, of course, that this technique does not generate a compiler; rather only the front end that handles lexical and syntactic issues. The point is that the development engineer's labor can be directed to semantic issues concerning what to do when a sentence in the language is recognized by the generated parser.

The great attraction of generative technologies is, of course, that no human labor need be spent on programming. The limitation of the technique is just as obvious: The approach is only applicable in those limited situations where the domain is so thoroughly understood and bounded that generation is feasible.

For domains that do not satisfy the rigorous requirements of generative technologies, an effective approach is the use of frameworks. An architecture-implementation framework is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style in code, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style. Frameworks may be extensive, requiring only minor portions to be completed by hand, or may be very basic, only offering help in implementing the most generic aspects of an architecture or architectural style. In any case, the places where hand work must be done are defined in advance. The developer need not, indeed must not, stray from doing work only in those locations, and must do the work in those locations in such a way that it does not violate the design of the framework or of the architecture.

The role of frameworks in the implementation of an architecture is placed in context in Figure 2-1. Framework selection and development is an integral part of architecture-based system implementation, because the framework provides the bridge between concepts from the architecture and concepts from the platform (that is, programming language and operating system). If a framework does not exist for a particular combination of architectural style

Figure 2-1. Frameworks and the implementation activity in the context of development. Activities are shown in the left column; artifacts are in the central column; notes are on the right.



and platform, conscientious developers will generally end up implementing one anyway because of the general software engineering principles of abstraction and modularity—frameworks encapsulate key features effective in mapping architectures to implementations.

As examples, common frameworks in extensive use are those targeted at supporting implementation of user interfaces. Both the Microsoft Foundation Classes (MFC) and the Java Swing library represent frameworks that provide the user with common user

interface widgets and interaction methods in object-oriented libraries. More than simply providing a set of facilities that users can take advantage of, these frameworks fundamentally influence the architectures of systems that employ them. For example, both frameworks (to some degree) manage concurrency and threading—the frameworks themselves create and manage the threads of control responsible for gathering user input events from the operating system and dispatching them to processing code developed by the user.

Related to frameworks, as aids for assisting in the implementation of architectures, are middleware technologies. Middleware comes in several varieties, some of which provide an extensive range of services, but as a whole they typically support communication between software components. As such, middleware facilitates the implementation of connectors—the architectural elements responsible for providing communication between components. The developer has complete responsibility for implementing an architecture's components' business logic, but relies on middleware for implementation of connectors.

Commonly found middleware includes CORBA and Microsoft's DCOM (for supporting communication between remote objects), RPC (for remote procedure calls), and message-oriented middleware (for delivery of asynchronous messages between remote components).

Generative technologies, frameworks, and middleware thus represent a spectrum of generic and widely used technologies for assisting in the faithful implementation of an architecture. These technologies progress from fully automatic, to partially automatic, to providing help with the implementation of connectors.

Less generic, but no less important, is the reuse of software components, whether commercial off-the-shelf, open-source, or in-house proprietary software. The key issue of such reuse, as it pertains to the faithful implementation of an architecture, is the degree to which the functionality, interfaces, and nonfunctional properties of a candidate component for reuse match those required by an architecture. In the (unfortunately) unlikely event that a perfect match exists, the job is done. In the common case, namely involving some degree of mismatch, the issues and choices are difficult. One strategy, which preserves the architecture as specified, is to encapsulate the preexisting component inside a new interface so that the thus-modified component exactly meets the requirements of the architecture. While this technique can be successful, a more likely circumstance is that performing such encapsulation still will not be economically or technically viable. In such cases, a decision needs to be made: whether to revise the architecture in such a way as to enable the reuse of the component, or to abandon this particular preexisting code and either search for alternatives or create a new component from scratch. This choice is a difficult one.

In absence of any technology to assist the developer in creating source code faithful to the architecture, the implementation must be developed manually. In such circumstances, the programmer must rely upon discipline and great care, for this can be the "great divide." To the extent that the implementation differs from the architecture that resulted from preceding development activities, that architecture does not characterize the application. The implementation *does* have an architecture: It is latent, as opposed to what is documented. Failure to recognize this distinction:

- Robs one of the ability to reason about the implemented application's architecture in the future.

- Misleads all stakeholders regarding what they believe they have as opposed to what they really have.
- Makes any development or evolution strategy that is based on the documented but inaccurate architecture doomed to failure.

Chapter 9 takes up these various issues in detail.

2.5 ANALYSIS AND TESTING

Analysis and testing are activities undertaken to assess the qualities of an artifact. In the traditional waterfall, analysis and testing are conducted after the code has been written, in the testing phase. In this context, the artifact being examined is the code; usually the quality being assessed is functional correctness, though properties such as performance may be examined as well. Analysis and testing activities do not have to be confined to occur after programming, of course, and many excellent development processes integrate these activities within the development activity. As long as an artifact exists that is susceptible to analysis for a defined property, that analysis can proceed—whether the artifact is a requirements document, a description of the application's structure, or something else. One of the virtues of conducting analysis before code exists is cost savings: The earlier an error is detected and corrected, the lower the aggregate cost.

Given that early detection of errors is a good thing, one must ask why conventional analysis and testing is almost always confined to simply testing source code. Moreover, why is such testing performed only against the most basic of specifications, namely the system functional requirements? The answer is almost always because of the nonexistence of any sufficiently rigorous representation of an application apart from source code. Rigorous representations are required so that precise questions can be asked and so that definitive answers can be obtained—preferably by means of automated analysis aids.

A technically based, architecture-centric approach to software development offers significant opportunity for early analysis and for improved analysis of source code. Additionally, the prospect for analyzing properties other than just functional correctness is present. In this approach to development, technically rich architectural models are present long before source code, and hence serve as the basis for and subject of early analysis. (Chapter 8 will present substantial details on architectural analysis, Chapter 12 will extend that discussion to non-functional properties, and Chapter 13 will delve in particular to architectural support for security and trustworthiness.) In the few paragraphs below we simply outline some of the approaches; details will come after our presentation of architectures and architectural models is on a more precise footing.

First, the structural architecture of an application can be examined for consistency, correctness, and exhibition of desired nonfunctional properties. If the architecture upon which the implementation is (later) based is of high quality, the prospects for a high-quality implementation are significantly raised.

As a formal artifact, the architectural model can be examined for internal consistency and correctness: Syntactic checks of the model can identify, for instance, mismatched components, incomplete specification of properties, and undesired communication patterns. Data flow analysis can be applied to determine definition/use mismatches, similar to how

such analysis is performed on source code. More significantly, flow analysis can be used to detect security flaws. Model-checking techniques can analyze for problems with deadlock. Estimates of the size of the final application can be based upon analysis of the architecture. Simulation techniques can be applied to perform some simple forms of dynamic analysis.

Second, the architectural model may be examined for consistency with requirements. Regardless of whether the requirements were developed in the classical way (that is, before any development of solution notions), or in the modern sense, in which the two are developed in concert, the two must be consistent. Such examination may be limited to manually performed analysis if the requirements are stated in natural language. Nevertheless, such comparisons are essential in guaranteeing that the two specifications are in agreement.

Third, the architectural model may be used in determining and supporting analysis and testing strategies applied to the source code. The architecture, of course, provides the design for the source code, so consistency between them is essential. Concretely, as the specification for the implementation, the architecture serves as the source of information for governing specification-based testing at all levels: unit, subsystem, and system level.

The architect can prioritize analysis and testing activities based upon the architecture, focusing activities on the most critical components and subassemblies. For example, in the development of a family of software products, special attention should be paid to those components that are part of every member of the product family: An error in one of them will affect all of the products. Conversely, a component that only implements a rarely used and noncritical feature of one member of the family may be judged to not merit nearly as much scrutiny. Testing costs money, and organizations must have a means for determining the priorities for their testing budget; architecture can provide the basis for some of that guidance.

In a similar vein, architecture provides a means for carrying forward analysis results from previous testing activities, with accompanying cost savings. For instance, if a particular component is reused in a new architecture, the degree of its unit testing can be reduced if examination of the architecture confirms that the context and conditions of use are the same as (or more constrained than) the earlier use.

Architecture can provide guidance and economies in the development of test harnesses. Test harnesses are small programs used to test components and subassemblies of larger applications. They provide the analyst with the ability to conveniently and effectively test the internal parts of an application. It can be difficult, for instance, to use system-level inputs to fully and economically exercise the boundary conditions of an internal component. For instance, if an architecture is designed to support a product family, and a small set of components are changed to customize the product for a particular market, a test harness can be constructed to focus testing on those components that comprise the change set. Each customized product can thus have a testing regimen emphasizing only those parts that have changed.

Architecture can also provide guidance in directing the analyst's attention to the connectors in a system's implementation. As will be discussed in Chapter 5, connectors play a particularly important role in some systems' structures, being tasked with supporting all intercomponent communication. The architecture provides an effective way for identifying those points of special leverage, and hence can aid in shaping the analysis and testing activity. Additionally, some connectors offer particularly effective opportunities for non-intrusive monitoring and logging of applications, which can play a key role in helping an

engineer develop an understanding of how a system works, as well as assisting in analysis and testing.

Fourth, the architectural model can be compared to a model derived from the source code of an application. This is a form of checking your answer. Stated abstractly, suppose program *P* is derived from architecture *A*. A separate team of engineers, with no access to *A*, could, by reading and analyzing *P*, develop an architectural model *A'*—a model of the architecture that *P* implements. If all is well, *A* will be consistent with *A'*. If not, either *P* does not faithfully implement *A*, or else *A'* does not faithfully reflect the architecture of *P*. Either way, an important inconsistency can be detected. This issue will be discussed in more detail in Chapter 3.

This abstract characterization is not very practical, as it would likely involve substantial human labor. But particular features of implementations can be readily extracted and compared with the architecture. For instance, it is straightforward to extract a model from source code that shows which components communicate with which others. That communication pattern can be compared to the corresponding communication pattern in the architecture and any discrepancies noted. An example of this type of analysis appears in the following chapter.

As noted above, substantive consideration of the analysis of architectures must wait until later in this text. Techniques for representing architectures must first be presented (see Chapter 6). It is sufficient for now to note that architecture-centric development provides a variety of new and significant opportunities for assessing and hence improving the quality of systems.

2.6 EVOLUTION AND MAINTENANCE

Software evolution and software maintenance—the terms are synonymous—refer to all manner of activities that chronologically follow the release of an application. These range from bug fixes to major additions of new functionality to creation of specialized versions of the application for specific markets or platforms.

The typical software engineering approach to maintenance is largely ad hoc. In a best-practices situation, each type of change causes the software process to return to whatever phase that issue is usually considered in, and the development process restarts from that point. Thus, if new functionality is requested, that requires returning to the requirements analysis phase and then moving forward in sequence from there. If the change is thought to be a minor bug fix, then perhaps only the coding phase and its successors are revisited.

The major risk presented by evolution is degradation of the quality of the application. Intellectual control of the application may degrade if changes may be made anywhere, by whatever means are most expedient. In practice, this is precisely what happens. Rather than following best practices, it is common for only the coding phase to be revisited. The code is modified in whatever way is easiest. Over time, the quality of the application degrades significantly, and making successive changes becomes increasingly difficult as complex dependencies between ill-considered earlier changes come to light.

An architecture-centric approach to development offers a solid basis for effective evolution. The key is a sustained focus on an explicit, substantive, modifiable, faithful architectural model.

The evolution process can be understood as consisting of the following stages:

- Motivation
- Evaluation or assessment
- Design and choice of approach
- Action, which includes preparation for the next round of adaptation

The motivations for evolution are many, as noted above. We draw special attention to the creation of new versions of a product. This motivation does not often appear in traditional treatments of software evolution, but is common in the commercial software industry. This kind of change raises the topic of software product families, and as discussed in Chapter 1, supporting product families can be a particular strength of architecture-centric approaches.

Whatever the motivation for evolution may be, the next step is assessment. The proposed change, as well as the existing application, must be examined to determine, for example, whether the desired change can be achieved and, if so, how. Put another way, this stage requires deep understanding of the existing product.

It is at this point that the architecture-centric approach emerges as a superior engineering strategy. If an explicit architectural model that is faithful to the implementation is available, then understanding and analysis can proceed efficiently. A good architectural model offers the basis for maintaining intellectual control of the application. If no architectural model is available, or if the model in existence is not consistent with the implementation, then the activity of understanding the application must proceed in a reverse-engineering fashion. That is, understanding the application will require examination of the source code and recovery of a model that provides adequate intellectual basis for determining how the needed changes can be made. This is time-consuming and costly, especially when the personnel involved are new to the project.

Errors in maintenance often arise from shortchanging this activity. If there is insufficient understanding of the existing structure, then plans to modify that structure will likely fail—at the points of misunderstanding. In contrast, a good architectural model offers a principled basis for deciding whether a desired modification is reasonable. A proposed change may be found to be too costly or detrimental to system properties to warrant development. Only with a good intellectual grip on the existing architecture, as well as the proposed change, can such a determination sensibly be made. The net effect of this principled approach will be the maintenance of architectural integrity and the attenuation of requirements volatility.

The next stage in evolution is development of an approach to satisfy whatever requirement motivated the activity. Several courses of action likely will be identified, so a further step of choosing among alternatives is required. Once a course of action is determined, it must be put into action. For changes of even moderate significance, the first artifact to be modified should be the model of the architecture. In particular, if the change needed pertains to the architecture, then changing the architecture is the place to begin. After changing the architecture corresponding changes to code can be made. The critical matter, of course, is maintaining consistency between the architecture and the implementation. Tools can help with this task.

A mistake to avoid is changing the code first and then planning to revise the architectural description to match. Following up afterward on that good intention seldom takes place.

Making the various changes needed to accommodate whatever motivated the evolution activity should not be considered complete until all elements of an application—architecture and code—are consistent. The reason is simple: Additional needs for evolution will be identified in the future. If the application is left inconsistent it will sabotage those future evolution activities.

Software will evolve, whether one is using an architecture-centric development process or not. The question is whether such evolution will proceed efficiently and whether it will result in the degradation of the quality and intellectual integrity of the product. Architecture provides the bedrock for maintenance of coherence, quality, and control. These issues are discussed in greater detail in Chapter 14.

2.7 PROCESSES

A key theme of this chapter has been that architecture concerns properly permeate the entire software development and evolution activity. The set of decisions comprising the architecture forms in concert with the requirements and continues to expand through maintenance. The architecture is thus under constant change.

Architecture, correspondingly, is not a phase in a software engineering process; it is a core, evolving body of information. Indeed, the development of this body of information may reach back and out to preceding applications and other applications.

The centrality of architecture to software development, and the insights that arise from a focus on architecture, are unfortunately totally obscured in traditional characterizations of the software development activity. Traditional software process discussions make the process activities the focal point; the architecture (and hence the product!) is often nowhere to be found. Equally bad, the standard software development processes encourage, if not enforce, sharp divisions between various types of development activities. For instance, the waterfall and spiral models both separate the activity of requirements determination from the activity of design development. As we have seen, such boundaries are not warranted and indeed are counterproductive.

Simply stating, however, that software development processes should be architecture-centric does not by any means say that there is a single correct way to proceed in application development. Different organizations and engineers will want to adopt particular strategies to best take advantage of their particular organizational strengths and preferences. Equally important, different development settings (for example, knowledge, preexisting developments, component libraries, codified architectures) will demand that different prominence be given to the various types of development activities, and those activities will vary in the amount of time required for their completion.

Comparing, or even understanding, different strategies for software development requires a means for describing those strategies. A good descriptive formalism for characterizing strategies will provide a way of not only showing what activities are occurring when, but will give appropriate and effective prominence to the central role of architecture (and all other project artifacts) in the formation of the software product. Accordingly, we

now present the *turbine visualization of software development*, and use it to portray a variety of development strategies.

2.7.1 The Turbine Visualization

The turbine visualization is a means for depicting an integrated set of software development activities in which the central role of software architecture in the evolution of the product can be prominently shown—if indeed it is central. The visualization accounts for the following independent aspects of software development:

- Time
- Kinds of activities active at any given time
- Effort (for example, labor hours expended) at any given time
- Product state (for example, total content of product development, or knowledge, at any given time).

The visualization also shows a variety of combined factors, such as investment (cumulative effort over time-demarcated phases).

Spatially, the visualization consists of a set of variable-width, variable-thickness rings stacked around a core. The axis of the core is time; the core represents the software product (the architecture plus all the other artifacts comprising the product); the rings represent time-demarcated phases of activity—all aspects of development or evolution.

The simple, unidirectional waterfall is as simple in the three-dimensional turbine visualization as it is in its traditional depiction—save that the role of the software product is now evident. A nominal waterfall process is shown in three-dimensional perspective in Figure 2-2. An annotated side view of the same process is illustrated in Figure 2-3.

The bottom ring of the waterfall turbine example begins with a null core, consists solely of the requirements analysis activity, and finishes at t_1 with a core consisting solely of the requirements document. The second ring begins at t_2 , consists solely of the design activity, and completes at t_3 , with the core now consisting of both the requirements document (unchanged since t_1) and the design document. The third ring consists solely of the coding activity, and the fourth solely of the testing activity. Each ring adds one more element to the core. The added value of the turbine model is that, by portraying the growing core, the set of existing product elements is made clear. The testing phase, for instance, visibly has reference to not only the subject of the tests (the code) but the requirements that the code is supposed to satisfy—including any acceptance tests developed during the requirements phase.

A cross-section shows the state of the project at a given point in time. The area of the ring is proportional to the number of labor hours currently being invested in the project. The area of the core indicates the content of the product at that same time. Subregions of the core show the relative development of different parts of the product. An example is shown in Figure 2-4, at time t_4 from Figure 2-3. The only activity is Design and the core consists of only the two documents shown.

The thickness of a ring denotes the time period during which the (possibly multiple, concurrent) activities of that ring are active, that is, its duration. Consequently, the volume of a ring (thickness multiplied by area) represents the investment made during that ring: the product of the time that ring was active and the investment made during

Figure 2-2.
The
unidirectional
waterfall model
as depicted by a
turbine, shown
from an angled
perspective.

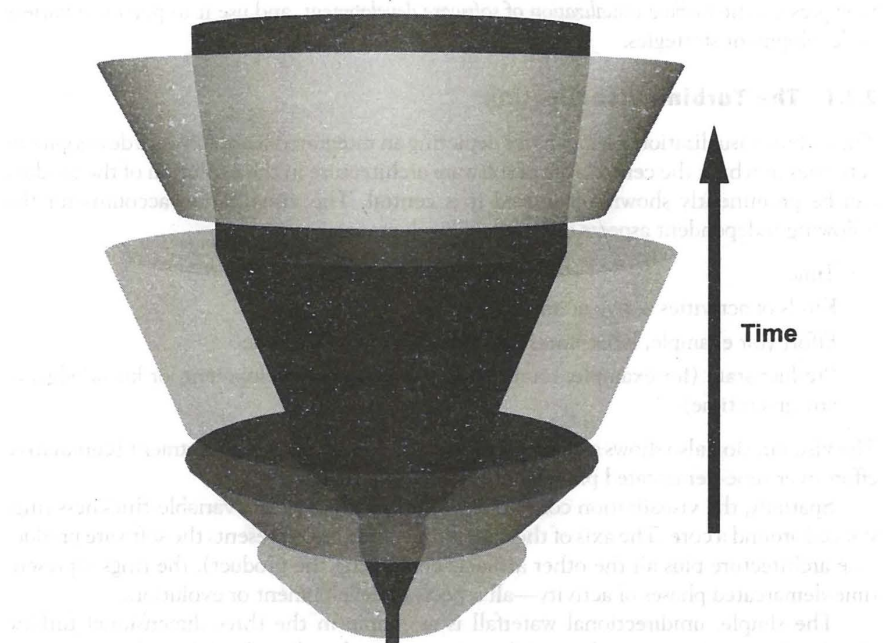
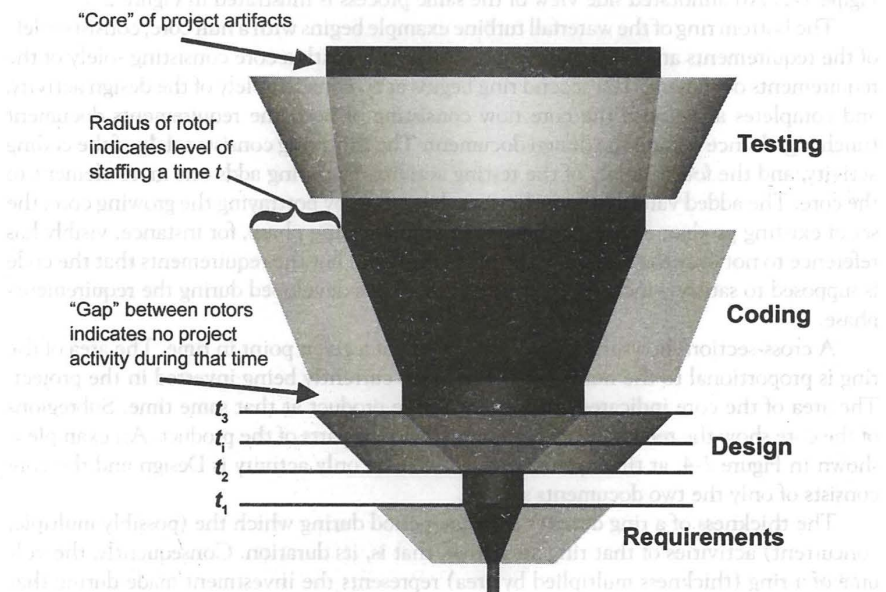


Figure 2-3.
Side view,
annotated, of the
simple process
depicted in
Figure 2-2.



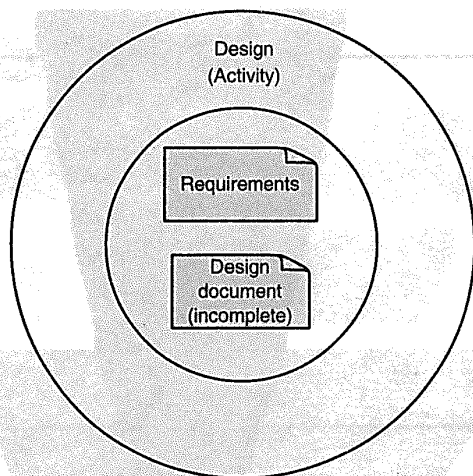


Figure 2-4.
Cross-section of
turbine from
Figure 2-3 at
time t_1 .

that ring. (Rings do not need to have the same cross-sectional area at t_1 as they do at time t_{i+1} , but we will make that simplifying assumption for the moment.)

Not all projects proceed in an uninterrupted fashion, so gaps between rings represent periods of inactivity in a project. During such gaps the core is nominally constant—though if project data gets lost, the core would shrink correspondingly. Rings may be divided into subareas, each of which represents a type of activity going on at that time. The activities shown in the subareas are *concurrent*. The relative size of the subareas denotes the share of the labor directed at that type of activity at that particular point in time.

A more complicated and more realistic illustration that involves multiple concurrent activities is shown in Figure 2-5; a side view for this nominal project is shown in Figure 2-6.

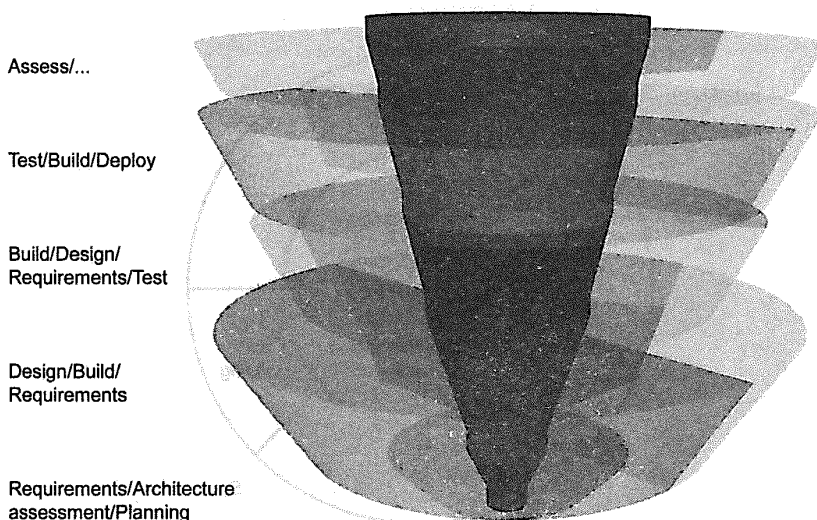
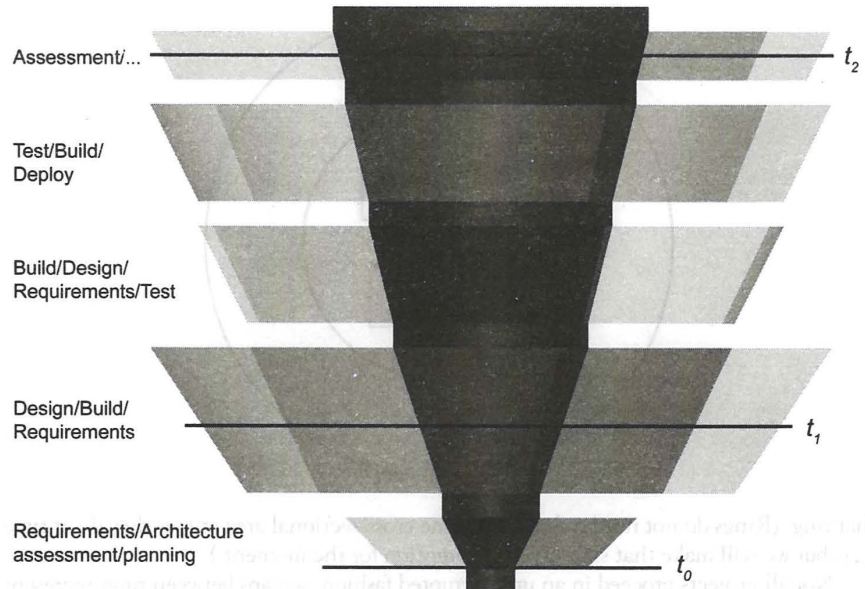


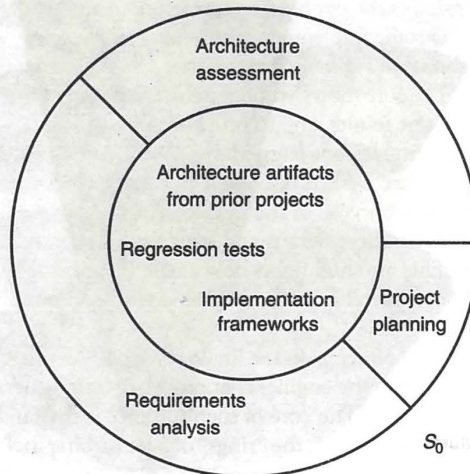
Figure 2-5.
Example turbine
visualization,
shown in a
three-dimensional
perspective view,
with rings shaded
by type of
activity.
Transparency is
used to enable
seeing activities
otherwise
obscured.

Figure 2-6.
Side view of the
project shown in
Figure 2-5.



Cross-section S_0 , shown in Figure 2-7, represents the state of the project at its beginning. The diameter of the core of the project is significant, representing substantial knowledge and resources carried forward from previous projects. The activities at time t_0 include architecture assessment (that is, evaluation of architectures from previous projects), requirements analysis, and project planning.

Figure 2-7.
Cross-section of
turbine at time t_0 .



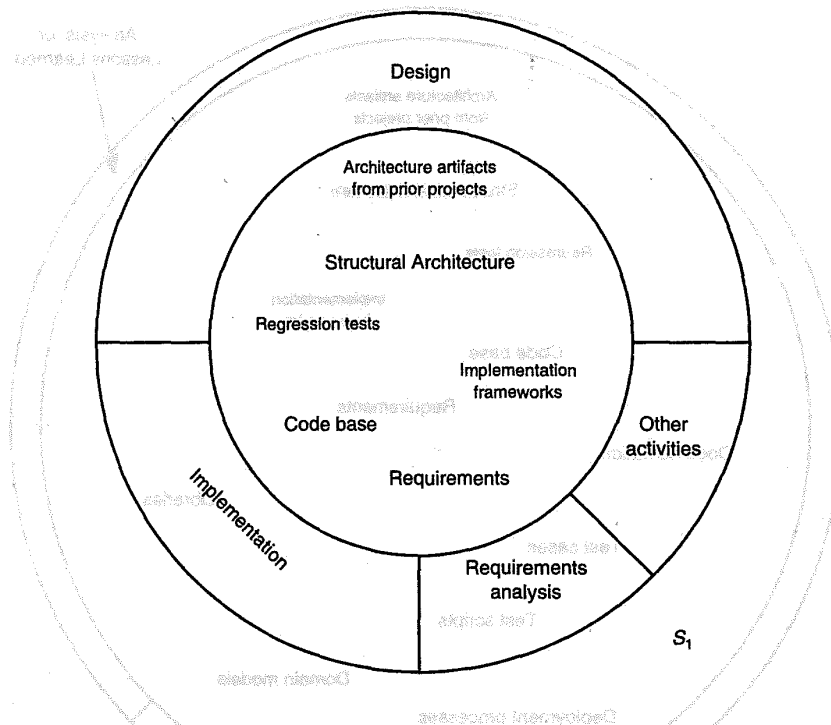


Figure 2-8.
Cross-section of
turbine at time t_1 .

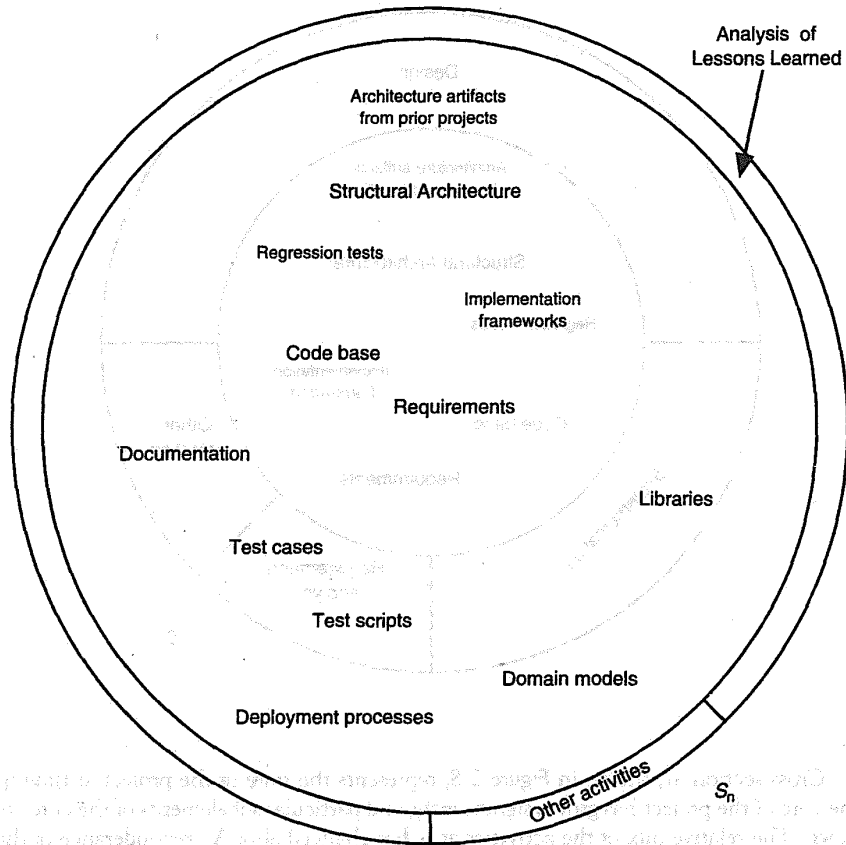
Cross-section S_1 , shown in Figure 2-8, represents the state of the project at time t_1 . The core of the project has grown significantly, and particular subelements of the core are shown. The relative mix of the activities at t_1 has changed also: A preponderance of the activity is devoted to design of the application's structure. A small amount of requirements analysis is still active, and some implementation is underway.

Cross-section S_n , shown in Figure 2-9, represents the state of the project near the time of its formal conclusion: The core shows a robust architecture, implementation, deployment processes, testing scripts, test results, and so on. The activity level is low, with remaining activities focused on capturing lessons learned.

Viewed as a whole (Figure 2-5), it is evident that the purpose of the rings is to build the core. In other words, the purpose of the development activities is to create the product, in all its various details. These details and structures are abundant at the end of the development activities. This also highlights how a wise organization must treat this core: It is a composite asset of considerable size and value, and should be managed, post-project, accordingly.

Before going on, a few comments on the analogy used in naming the visualization are in order. Turbines are, of course, the engines that propel aircraft, generate hydroelectricity, and power myriad other devices. The core of such turbines is the fluid that flows through them. With jet turbines the fluid is air; the "rings" of a jet turbine include the compressors with their fan blades whose job it is to compress the air, pushing it into the combustion

Figure 2-9.
Cross-section
of turbine at
time t_n .



chamber, where fuel is added and ignited. Other rings behind the combustion chamber contribute to the overall process, in which the air, now in large volume and at high speed, exits the engine.

The analogy to software development is only a mild one and should not be pressed very far. But the product is the focus of the turbine; each ring and element contributes to that product. The air flows through the turbine, and is touched by each of the various rings. So, too, does the software product and the architecture, in particular, represents the core of software development. It is touched and (it is hoped) enhanced by each aspect of software development. In contrast to a physical turbine, however, each ring of software development genuinely adds to the core, not just heating it up. (Other difficulties—or advantages—of the analogy, such as the rings spinning in circles and the product just being hot air, are strictly unauthorized!)

2.7.2 Example Process Depictions

Two further examples are shown now, where use of the turbine visualization technique highlights distinctive aspects of these approaches to software development.

A Robust Domain-Specific Software Architecture-based Project

Figure 2-10 shows a turbine visualization of a notional project that was based upon a preexisting domain-specific software architecture, that is, a project similar to the Philips television example of Chapter 1. At the project outset, the core is quite large; it contains a multitude of reusable artifacts from preceding projects. The activities are thus (1) to assess the artifacts to ensure that the current project fits within the constraints imposed by those artifacts, (2) to parameterize those artifacts to meet the new project's needs and to perform any customization necessary, and (3) to integrate, assess, and deploy the product.

Agile Development

The turbine visualization can be used in analyzing alternative approaches to software development. For example, Figure 2-11 shows a notional agile development process. Agile processes are positive in showing, and emphasizing, concurrency between a variety of kinds of development activities; requirements elicitation and development of tests, for instance. As the development progresses those activities do not cease, for example, once code development is begun. Indeed all of these activities continue throughout the project.

As the skinny core of the turbine model indicates, however, the agile process denies development of any explicit architecture; rather for the agile developer the code is the architecture. The visualization indicates that the agile process starts with a core that is devoid of any architecture and terminates similarly. A large body of code may be present, along with, perhaps, requirements documents or user guides, but no explicit record of the fundamental design decisions, such as the application's architectural style.

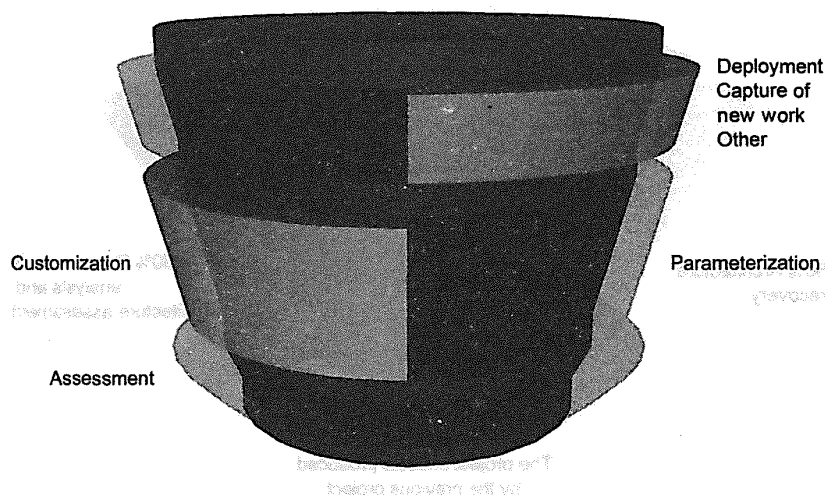
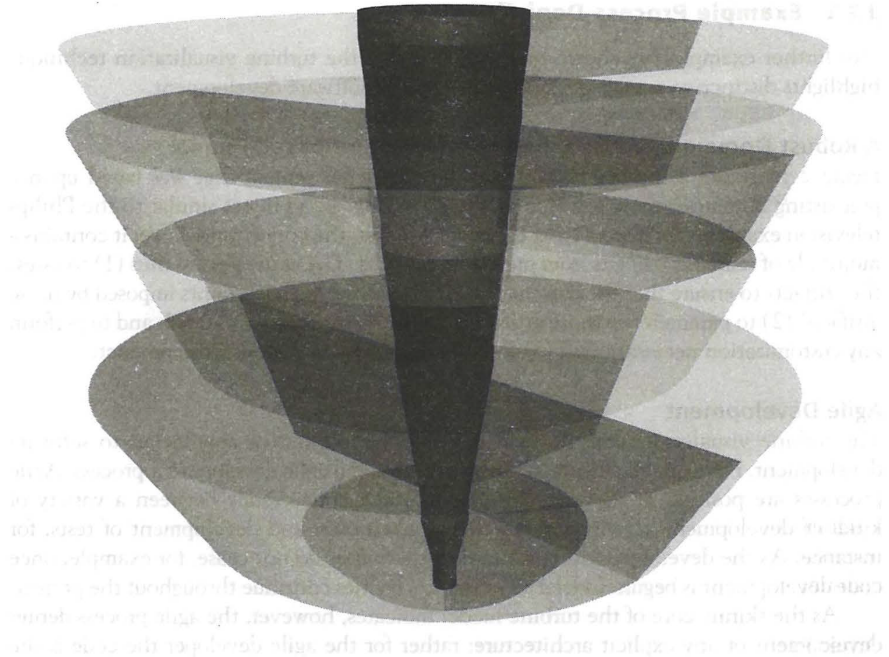


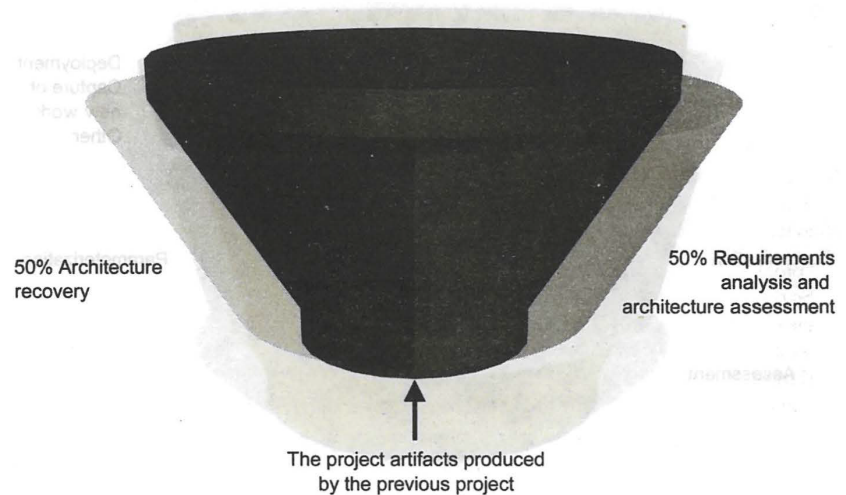
Figure 2-10.
A development
process based on
an existing
domain-specific
software
architecture and
reuse library.

Figure 2-11.
Turbine
visualization of a
notional agile
development
process.



The problems with this approach are made clear when a follow-on project is required. That is, if at some point after the first agile project completes its development, a follow-on project is required to have the application meet some new demand. Unless the same development team—with excellent memories—is employed on the subsequent project, an initial project model such as shown in Figure 2-12 may be anticipated. In particular,

Figure 2-12.
Initial portion of
a turbine model
of a phase 2 agile
process.



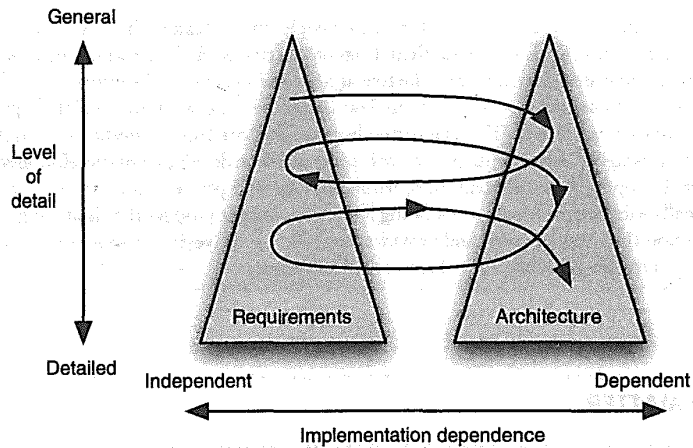


Figure 2-13.
Twin Peaks
model of
development.
© IEEE 2001.

a significant ring of activity will be required at the beginning of the project to simply understand the existing code base and partially recover the latent architecture so that planning for how to meet the new needs can proceed.

Other Processes and Process Models

A variety of researchers and organizations have promulgated processes that, to a greater or lesser extent, show the special role of architecture in development. One of the best of these is Professor Bashar Nuseibeh's Twin Peaks model Nuseibeh (Nuseibeh 2001). Twin Peaks emphasizes the co-development of requirements and architectures, incrementally elaborating details. The model is illustrated in Figure 2-13, which is derived from the Twin Peaks paper.

It is left as an exercise for the reader to create a turbine visualization of the Twin Peaks process shown in the figure. It should also be noted that the Twin Peaks work is representative of recent work in requirements engineering that is now giving much more prominence to the role of design and existing architectures in the activity of product conception.

"Brooks' law: Adding people to a late software project makes it later."

This adage is one of the best known in software engineering. Yet the reasons why it is so often true are seldom explained. Fred Brooks gave those reasons in an interview:

"Brooks' law depends heavily on the amount of information that has to be communicated. So the argument is that if you add people to a project that you already know is late, which means you're at least in the middle of the project, you have to repartition the work. That's a job in itself: Just deciding who is going to do what means that instead of having the thing divided into the units you had it divided into, you have to divide it into more units. Sometimes that can be done by subdividing the existing units, but sometimes you have to move boundaries."

(Fred Brooks, quoted in *Fortune*, December 12, 2005.)

**Brooks'
Law and
Software
Architecture**

Additional impacts include training of the new people on the team—that takes the time and energy of the existing staff, removing them from productive work. Then when the new people start working they are relatively unproductive at first, and likely to make errors.

Note that Brooks did not say that his law is true because we have immature processes or poor management ability. The law is true because the intellectual substance of software is profound and weighty. An architecture-centric perspective on development therefore gives some advantage in coping with problems in schedule slippage and project management: It provides a technically substantive basis for reasoning about the nascent product that is at a higher level of abstraction than source code, and provides a vehicle for conveying to new project members the principal design decisions that characterize the system.

2.8 END MATTER

This chapter has shown how a proper view of software architecture affects every aspect of the classical software engineering activities and reorients those activities. Any practitioner can have his work informed and changed by an understanding of software architecture, to the project's benefit. In summary,

- The requirements activity is seen as a co-equal partner with design activities, wherein previous products and designs provide the vocabulary for articulating new requirements, and wherein new design insights provide the inspiration for detailed product strategies and requirements.
- The design activity is enriched by techniques that exploit knowledge gained in previous product developments. Designing infuses the entire development and evolution process, instead of being confined to a stand-alone activity.
- The implementation activity is centered on creating a faithful implementation of the architecture and utilizes a variety of techniques to achieve this in a cost-effective manner, ranging from generation of the source code to utilizing implementation frameworks to reuse of preexisting code.
- Analysis and testing activities can be focused on and guided by the architecture, offering the prospect of earlier detection of errors and more efficient and effective examinations of the product. Higher quality at a lower price should be the result.
- Evolution activities revolve around the conceptual bedrock of the product's architecture. Critically, it provides the means for reasoning about possible changes and for conveying essential understandings of the product to engineers who are new to the project.
- An equal focus on process and product results from a proper understanding of the role of software architecture in the development activity. The turbine visualization enables insight into a development process's full character, as it can reveal the extent to which activities are intermingled (likely a very good thing) and especially the extent to which the corpus of project artifacts changes over time.

Despite these insights, attempting to appropriate and apply these ideas without further detail and technical support is difficult and subject to the limitations of a designer's

organization and self-discipline. It is the purpose of the following chapters to provide the representational basis for describing architectures effectively and the technological basis for incorporating architecture into a professional approach to the development of software applications. Chapter 3 proceeds to put the concepts on a firm definitional footing; Chapter 4 then proceeds to describe how to design systems from an architectural perspective.

Sales of a product provide revenue for a company. For a company to grow, sales typically must expand through offering a range of products. For a company to retain its customer base it must be responsive to requests for altered versions of current products, as well as new offerings.

While these observations are almost trivial, the question arises as to what enables a company to perform these tasks successfully over an extended period. Again, the almost trivial answer is a sustained focus on the company's processes as well as a focus on the company's products.

What is remarkable is that over the past twenty years or so this dual focus seems to have been lost in a large number of software organizations. For many, the focus has been solely a focus on process and its improvement. The results of this myopia have been rather mixed. Clearly, there have been many examples of companies improving their processes and also being successful in producing high-quality, successful products. But there are also companies—much less publicized—that have achieved high levels of process improvement but that fail to produce successful, market-leading products. The correlation between process improvement and product success is not absolute, to say the least.

In contrast, consider the relationship between a product's architecture and its quality. It is difficult to have a bad architecture and great product, a product that remains a leader after successive versions and releases. Similarly, it is difficult to have a great architecture and a bad product. Unless the implementation activity is deeply flawed, a great architecture—one that emerges from design that considers all stakeholder concerns and that exhibits intellectual clarity and vision—typically will result in great products.

The correlation between a great architecture and a successful product family is perhaps even stronger. A cost-effective strategy for delivering a good product family demands intellectual coherence across the members of the product family and cost-savings resulting from commonalities and shared infrastructure across the line.

An architecture-centric approach to software development thus puts primacy on the products that are sold and that provide a company's revenue. If process is allowed to become the primary focus, then the focus is on something that does not intrinsically generate revenue for the company. A good process is an asset—a critical one at that—but architecture, at the heart of great products, must be of primary attention.

The Business Case for Architecture-centric Development

2.9 REVIEW QUESTIONS

1. Does object-oriented design impose an architecture on an application? Why or why not?
2. Is an architecture-centric viewpoint more consistent with the waterfall model or Boehm's spiral model of software development (Boehm 1988)? Why?
3. Suppose a feature-addition request is given to a development team that has not previously worked on the product in question. How might the team proceed if it is not in possession of the product's architecture? If it is in possession of an accurate, rich architecture?

2.10 EXERCISES

1. For some application that you have developed, describe its architecture, as built. Is that the same as the design you originally developed for the application? What accounts for any differences you observed?
2. Describe an application for which you can develop a complete, adequate requirements document without any recourse to thinking about how the system might be built. Describe an application for which you think it would be difficult, if not impossible, to describe the requirements without recourse to architectural ("how to") concerns.
3. Investigate the history of an influential software product, such as spreadsheets (for example, VisiCalc, Lotus 1-2-3, or Excel), or a service product (for example, eBay or Google). What role did requirements analysis play in the development of the product? How early were design prototypes created?
4. Develop three alternative architectures for a sample problem (such as a simple video game like Tetris or an address book). What makes them different and how did you arrive at the architectures?
5. Consider the application domain of video games, in particular very simple games based on the idea of landing a spacecraft on the moon by controlling the amount of descent thrust. Create a glossary for use in requirements descriptions, including such key terms as descent rate, controls, display, and score. Do notions of solution structure affect development of the definitions? If so, how?
6. Do Exercise 5, but for the domain of spreadsheets. Make sure your glossary includes all the terms needed to fully characterize the spreadsheet's concepts.
7. How does a company build market niche? How could that affect the company's approach to software processes?
8. Is it easier to build a parser for a programming language by manually programming in an object-oriented language, such as Java, or use a parser generator, such as ANTLR or Bison? What assumptions does your analysis make?
9. With regard to Exercise 8, what does implementing a parser using a parser generator entail? Where did the architecture of the (generated) parser come from?
10. With regard to Exercises 8 and 9, if you were responsible for testing both a manually written parser and a parser generated by a parser tool, how would your testing concerns and strategies vary between the two products? Why?
11. Abstract requirements may work on small problems wherein the engineer can, after specification, devise many solutions quickly, or there may only be a small number of solutions, *and* the total time involved in finding one solution starting from scratch is not so great as to matter. But above a certain threshold, it appears that it becomes either economically inefficient to pursue this course of action or it is intellectually too difficult to pursue this practice, or both. Describe a small domain where you illustrate this point. In which domains do you think that the use of abstract requirements have been most effective? To what do you attribute the success?
12. The chapter described how the width of a rotor in the turbine visualization should be proportional to the amount of labor directed at the development activity at a given point in time. For example, the width could be a function of the number of labor hours per day devoted to development. What should the width of the turbine's core be? It should be a measure of the information space of the project. Consider the relative merits of measuring: the number of megabytes of project artifacts, the number of project artifact files, and the number of source lines of code or text documents. What other measures could be used?
13. Draw a turbine visualization of a notional Twin Peaks software development process.
14. Draw a turbine visualization of a notional software project that follows the Unified Process [see (Kruchten 2000; Larman 2002; Schach 2007) for example explications].

2.11 FURTHER READING

Substantive presentations of software engineering and software development processes can be found in any of several excellent textbooks, such as those by Ghezzi et al. (Ghezzi, Jazayeri, and Mandrioli 2003), and van Vliet (van Vliet 2000).

Problem solving and design in general has been addressed by many authors. Design is the focus of Chapter 4, and a variety of readings are listed at the end of that chapter. Polya's original book on problem solving—which largely focuses on the solving of problems in mathematics—is readily available (Polya 1957). Some general reflections on design and the design process include Schön's classic *The Reflective Practitioner* (Schön 1983), Don Norman's insightful book on *The Design of Everyday Things* (Norman 2002), and Nelson's treatment of the traditions and practices of design (Nelson and Stolterman 2003). For the engineer in all of us, Henry Petroski's books are a delight, exploring the design of things from bridges to paperclips, but they are especially valuable in revealing the role of failure in achieving new designs (Petroski 1985, 1992, 1994, 1996). Several detailed stories of how design of the F-15, F-16, and A-10 aircraft were, or nearly were,

subverted due to loss of focus on the core architecture are told in *Boyd: The Fighter Pilot Who Changed the Art of War* (Coram 2002). The history of the design of washing machines, mentioned briefly in the text, is presented online in a variety of Web sites, including www.sciencetech.technomuses.ca/english/collection/wash1.cfm and www.historychannel.com/exhibits/hometech/wash.html.

Excellent presentation of contemporary thinking on the process of requirements analysis can be found in the works of Jackson, van Lamsweerde, and Nuseibeh (Jackson 1995; Jackson 2001; Nuseibeh and Easterbrook 2000; van Lamsweerde 2000). Object-oriented design is explained in most modern software engineering textbooks, such as those listed above. Software analysis and testing is explained in detail in Pezzè and Young's textbook (Pezzè and Young 2007).

The Twin Peaks model of development was presented in a short paper in *IEEE Computer* (Nuseibeh 2001). Subsequent work has explored more deeply the relationship between requirements and architecture; see for example (Rapanotti et al. 2004).

Basic Concepts

The preceding chapters informally discussed a number of software architectural notions, such as software components and connectors; their configurations in a given system; relationships between a system's requirements, its architecture, and its implementation; and software product lines. These ideas served to provide context for the field of software architecture, situate it within other facets of software engineering, and motivate its unique role and importance. Indeed, we have generally avoided definitions of terms in the hope that the reader will more readily recognize many of the discussed concepts within his own experience, or be able to relate the architectural concepts to those with which he is already familiar.

While informal terms have a useful role in introducing concepts, the uncertainties that result can hinder deeper understanding. Hence, the objective of this chapter is to define the key terms and ideas from the field of software architecture, providing a uniform basis for their discussion in the remainder of the book. Key elements of architecture-centric design and their interrelationships, basic techniques and processes for developing a software system's architecture, and the relevant stakeholders and their roles in architecture-based software development are presented and illustrated with simple examples.

3 Basic Concepts

3.1 Terminology

- 3.1.1 Architecture
- 3.1.2 Component
- 3.1.3 Connector
- 3.1.4 Configuration
- 3.1.5 Architectural Style
- 3.1.6 Architectural Pattern

3.2 Models

- 3.3 Processes
- 3.4 Stakeholders
- 3.5 End Matter
- 3.6 Review Questions
- 3.7 Exercises
- 3.8 Further Reading

Outline of Chapter 3