Declaration of Catherine Vassilkova

1. I, Catherine Vassilkova, am an employee of BEADOX LLC ("BEA").
2. I am over the age of 21 and have personal knowledge of the facts contained herein unless otherwise indicated.
3. I have worked in the field of information research and document retrieval for law firms since 2005.
4. I have been with BEA since I established the company in 2019.
5. BEA, based in Silver Spring, MD, conducts research at the District of Columbia federal and local courts, archives, government agencies, and libraries around the world. BEA's services include interlibrary loans, specialized research, article location and delivery, and other document retrieval. One of my responsibilities is to scan textbooks for requesters.
6. BEA specializes in the research and delivery of materials from the Library of Congress.
7. On June 7, 2024, I was asked by the requester Morrison and Foerster LLP, through the University of Wisconsin, to scan the following textbook: Software architecture : foundations, theory, and practice (2010).
8. On June 7, 2024, I obtained a copy of the textbook from the Library of Congress and scanned the requested excerpts including book cover, title page, copyright page date-stamped (March 11, 2009), table of contents pages and pages 91-139 & 574-606. Those excerpts are attached as Attachment A.
9. I confirm that the copyright page of the scanned textbook had a stamped date of receipt of March 11, 2009.
10. I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and that these statements were made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under section 1001 of Title 18 of the United States Code.
11. Executed the 9th of June 2024 in Silver Spring, MD 20910.

Signed:



Attachment A

Why is the World Wide Web so successful? What has enabled it to scale across millions of servers and untold users? How can a company leverage its corporate knowledge to produce economically a family of software products that dominate a market segment? At the core of the answers to these questions is software architecture, the set of principal design decisions made about a system. This text reveals the fundamentals of software architecture, and enables the reader to employ the latest techniques to produce software applications that have conceptual integrity and meet all stakeholders' expectations.

Incorporating the latest research results, this book:

- Shows how a proper focus on software architecture transforms the software development process
- Teaches how to design the architecture of a system, including techniques appropriate for established application areas for which abundant experience exists
- Presents methods for designing systems to meet non-functional requirements
- Focuses on developing implementations that are faithful to their architecture, and which are adaptable to meet future needs
- Contains numerous examples from practice
- Surveys and analyzes numerous modeling techniques, standards, and practices

Written by internationally-renowned experts with both industry and academic experience, this book's ambitious scope makes it a suitable and unique reference for software engineering and software architecture students, researchers, and practitioners alike.

Richard N. Taylor is a Professor of Information and Computer Sciences at the University of California, Irvine. He received the Ph.D. degree in Computer Science from the University of Colorado at Boulder in 1980. Professor Taylor is the 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.

Nenad Medvidović is Director of the University of Southern California's Center for Systems and Software Engineering, and an Associate Professor in the Computer Science Department at USC. Medvidović received his Ph.D. in 1999 from the Department of Information and Computer Science at University of California, Irvine. Medvidović is a recipient of the National Science Foundation CAREER award (2000), the Okawa Foundation Research Grant (2005), and the IBM Real-Time Innovation Award (2007).

Eric M. Dashofy is a Senior Member of the Technical Staff at The Aerospace Corporation in El Segundo, CA. He received his Ph.D. in 2007 from the University of California, Irvine. He is the lead developer of the ArchStudio architecture-centric software development environment, an open-source, Eclipse-based toolset for modeling, visualizing, analyzing, and implementing architecture-based software systems.

WILEY
www.wiley.com/college/taylor

LC CIP
0 023 972 772 4

ISBN 978-0-470-16774-8
90000
9 780470 167748

QA 76
.754
.T39
2010
Copy 1

Taylor
Medvidović
Dashofy

SOFTWARE ARCHITECTURE
FOUNDATIONS, THEORY, AND PRACTICE

SOFTWARE ARCHITECTURE

FOUNDATIONS, THEORY, AND PRACTICE



Richard N. Taylor • Nenad Medvidović • Eric M. Dashofy

Sony Exhibit 1026 Page 3
Sony v. Quantum IPR2023-00954

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

John Wiley & Sons, Inc.

Sony Exhibit 1026 Page 4
Sony v. Quantum IPR2023-00954

LIBRARY OF CONGRESS
MAR 11 2009
CIP

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

To Pavle and Sasha
—*Nenad Medvidović*

To Mom and Keith
—*Eric M. Dashofy*

LC Control Number
2009 280435

Vice President & Publisher	Donald Fowley
Acquisitions Editor	Daniel Sayre
Senior Editorial Assistant	Carolyn Weisman
Executive Marketing Manager	Christopher Ruel
Senior Production Editor	Sandra Dumas
Production Manager	Dorothy Sinclair
Design Director	Jeof Vita
Designer	Lee Goldstein
Cover Designer	James O'Shea
Senior Media Editor	Lauren Sapira
Production Management Services	Elm Street Publishing Services

This book was set in 10/12 Goudy by Thomson Digital and printed and bound by R. R. Donnelley/Crawfordsville. The cover was printed by R. R. Donnelley/Crawfordsville.

This book is printed on acid free paper. ∞

Copyright © 2010 John Wiley & Sons, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008, website http://www.wiley.com/go/permissions.

To order books or for customer service please, call 1-800-CALL WILEY (225-5945).

ISBN-13 978-0470-16774-8
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

LIBRARY OF CONGRESS
MAR 11 2009
CIP

SOFTWARE ARCHITECTURE

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

Foundations, Theory, and Practice

To Pavle and Sasha
—*Nenad Medvidović*

LC Control Number
2009 280435

To Mom and Keith
—*Eric M. Dashofy*

Richard N. Taylor
University of California, Irvine

Nenad Medvidović
University of Southern California

Eric M. Dashofy
The Aerospace Corporation

Vice President & Publisher	Donald Fowley
Acquisitions Editor	Daniel Sayre
Senior Editorial Assistant	Carolyn Weisman
Executive Marketing Manager	Christopher Ruel
Senior Production Editor	Sandra Dumas
Production Manager	Dorothy Sinclair
Design Director	Jeof Vita
Designer	Lee Goldstein
Cover Designer	James O'Shea
Senior Media Editor	Lauren Sapira
Production Management Services	Elm Street Publishing Services

This book was set in 10/12 Goudy by Thomson Digital and printed and bound by R. R. Donnelley/Crawfordsville. The cover was printed by R. R. Donnelley/Crawfordsville.

This book is printed on acid free paper. ∞

Copyright © 2010 John Wiley & Sons, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008, website http://www.wiley.com/go/permissions.

To order books or for customer service please, call 1-800-CALL WILEY (225-5945).



ISBN-13 978-0470-16774-8
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

John Wiley & Sons, 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

Sony Exhibit 1026 Page 7
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 8
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 9
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 10
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 11
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 12
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 13
Sony v. Quantum IPR2023-00954

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

Sony Exhibit 1026 Page 14
Sony v. Quantum IPR2023-00954

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**

Sony Exhibit 1026 Page 15
Sony v. Quantum IPR2023-00954

prior work need not only come from one's own experience, of course. One virtue of reading widely in a technical field or attending technical conferences is to benefit from the experiences of others. There is no virtue in reinventing wheels, broken or otherwise.

When valuable technical experience is possessed, however, substantial efficiencies result. Experience can provide that initial feasible set of "alternative arrangements for the design as a whole" called for at the beginning of Section 4.1. One need not devote valuable design time to topics for which one already has a validated solution in hand. Similarly, communication between development team members is enhanced: when others have experience with the same or similar preexisting systems, it is easier to describe use of or modifications to an established approach, as compared to conveying a completely new concept.

Applying established approaches is not an entirely risk-free strategy however. Once one has a favorite hammer, everything starts to look like a nail. Pursuing such a strategy may yield a working application, but can easily yield one that is seriously deficient in some respect or is at least suboptimal. Not only are the benefits of previous systems imported by reusing an approach, but so are the limitations. Innovation is certainly inhibited by mindlessly following one's standard approach. That said, it is nonetheless clear that refined experience is indeed the Grand Tool for designers to apply in creating new applications.

Silver Bullets and Other Misfires

Software engineering literature is replete with surefire approaches to design—tools and techniques that are assured to always work and provide that needed solution. Some of these mistaken approaches to design confuse a notation or language with a design method. UML, for instance, is a notation, not an approach. So, too, is Java. Other approaches confuse a tool or technology with a design method. Thus some have said that CORBA provides all the needed guidance for building robotic systems; others have assured us that rule-based expert systems are all we need. Most assuredly, there is no magic potion, and an adequate approach to design problems demands care, thought, creativity, discipline, experience, and method.

The software designer of the twenty-first century has a wide basis of experience from which to draw. Reflection on the development of thousands of systems has yielded a wide variety of approaches and lessons, ranging from broadly encompassing domain-specific architectures to simple programming language-level design patterns. The following section presents key lessons from this experience base in considerable detail.

4.3 REFINED EXPERIENCE IN ACTION: STYLES AND ARCHITECTURAL PATTERNS

The distilled architectural wisdom that has arisen from a vast body of experience comes in many shapes and flavors. As a whole, we are treating them under the rubric of styles and patterns. As introduced in Chapter 1, the notion of architectural styles is an ancient one, dating back at least to Vitruvius. Styles are designed to capture knowledge of effective designs for achieving specified goals within a particular application context. For building architecture, the combination of goals and context might be providing a house for a single family in a Mediterranean climate using stone and tile as principal building materials. The

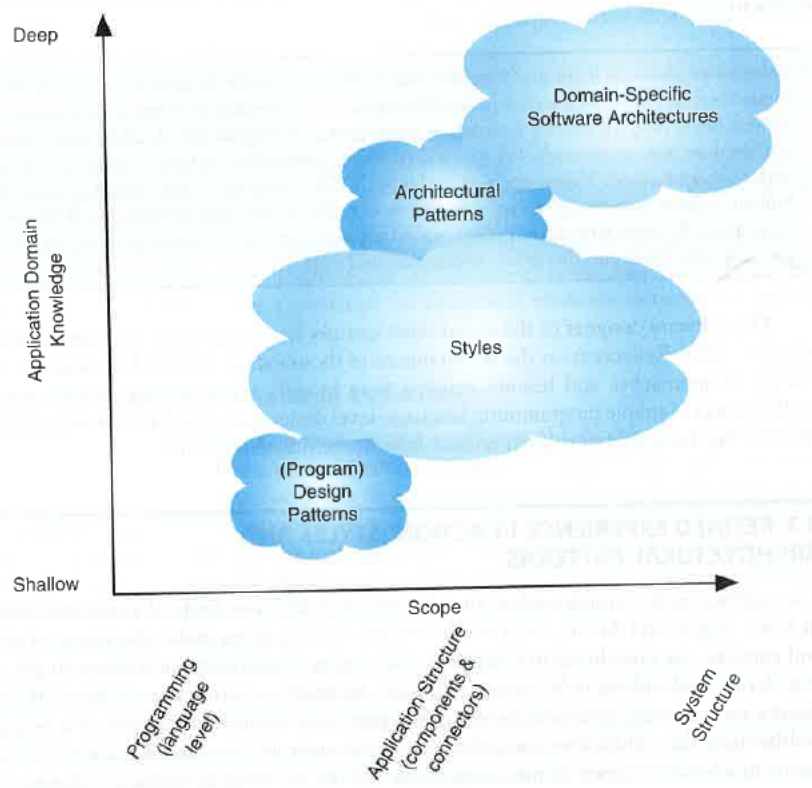
Sony Exhibit 1026 Page 16
Sony v. Quantum IPR2023-00954

correspondingly appropriate style might be a single-story villa. For software, the goals and context might be providing a secure instant messaging system to operate between remote sites of a global company; the appropriate style might be client-server.

Patterns and styles can be characterized for large problems as well as small. Again, with regard to buildings, one can characterize a pattern for determining the appropriate overhang of a tiled roof over an entryway; similarly, one can characterize a (much more complicated) pattern for provision of elevator service in a ninety-story skyscraper. So it is with software architecture; patterns and styles have been developed that are appropriate for small as well as large and complex applications.

Not surprisingly, the lessons of experience in software architecture have arisen from several different software development communities, with each having its own contribution to make. Unfortunately, this has resulted in a variety of terms for characterizing this experience, not all of which describe distinct concepts. As an aid to teasing out the various issues and as a means to provide structure for the rest of this section, see Figure 4-2 for a summary of the main types of architectural styles and patterns.

The horizontal axis of Figure 4-2 refers to the scope of applicability of the body of knowledge—the domain of discourse. For instance, design patterns exemplified by those



presented in the book by Gamma (Gamma et al. 1995), are focused on the programming of object-oriented solutions to program design problems. Such patterns are inapplicable to enterprise-level system design, which is at the other end of the horizontal axis.

The vertical axis reflects the amount of domain knowledge represented by (or encoded within) the body of knowledge. Program design patterns are shown at the shallow end of this scale, reflecting that they are very generally applicable and are not restricted to use within a particular application domain. At the other end of the scale are domain-specific software architectures. Such styles are applicable to the design of very large applications and concomitantly encode substantial knowledge about the design of applications within a domain. Thus, for instance, a domain-specific software architecture for retail banking might specify all major parts of the application; it is unlikely, however, that such a structure would be appropriate for any other kind of application apart from retail banking.

The diagram has fuzzy boundaries, however, and should only be taken as a rough guide. Programming languages exist at many levels of abstraction; applications range from very simple to extraordinarily complex. The scope axis should not, therefore, be taken as genuinely linear or fully ordered. The concept clouds in the diagram similarly have fuzzy boundaries; what one architect may term an architectural pattern might be called a style by another. The following sections will bring a little more precision to the discussion. While the topic of domain-specific architectures was introduced in Chapter 2 and will be dealt with at substantial length in Chapter 15, we begin the following exposition by a brief consideration of such architectures, as they are a powerful way of exploiting experience in the task of developing the architecture for a new system. We then consider architectural patterns and styles at length. These topics were introduced and defined in the preceding chapters; here, a wide variety of specific instances are presented. This chapter does not include any further discussion of program design patterns. As Figure 4-2 indicates, such patterns deal with concepts at the level of programming languages, and most often only for object-oriented languages. As such, they offer little guidance to the architect seeking to identify a feasible set of "alternative arrangements for the design as a whole." (They do, of course, offer significant guidance to a programmer charged with implementing a chosen architecture in an object-oriented language, but that is not the topic of this chapter.)

4.3.1 Domain-Specific Software Architectures

Domain-specific software architectures (DSSAs) encode substantial knowledge, acquired through extensive experience, about how to structure complete applications within a particular domain. A useful operational definition of DSSAs is the combination of (1) a *reference architecture* for an application domain (as defined in Chapter 3), (2) a library of software components for that architecture containing reusable chunks of domain expertise, and (3) a method of choosing and configuring components to work within an instance of the reference architecture (Hayes-Roth et al. 1995). As such, a DSSA represents the most valuable type of experience useful in identifying a feasible set of "alternative arrangements for the design as a whole."

In terms of our familiar building analogy, DSSAs are akin to the generic design of tract houses, that is, house designs that are instantiated dozens or hundreds of times to yield individual homes in a housing development. Such homes are not identical to one

Sony Exhibit 1026 Page 18
Sony v. Quantum IPR2023-00954

another; they typically vary in terms of cabinetry, banister styles, colors, carpet, roofing material, and such. They will be same, though, with respect to the floor plan, the location of the windows, and with regard to all major structural elements. The contractor for an individual house knows how to instantiate the generic plan in light of the consumer's specific preferences and choices to yield a very cost-effective structure that nonetheless is somewhat personalized.

The extended example of software for consumer electronics presented in Chapter 1 serves as an illustration of a DSSA. (While this example was presented there as an illustration of product families, the two concepts are closely related; this relationship is carefully explored in Chapter 15.) Different consumer electronics applications can be created by parameterizing or specializing the core architecture. These specializations may represent adding new, unprecedented components into a television's architecture, for instance, but where those additions are performed on parts of the architecture are identified in advance as places for such extensions to occur.

The difficulty with DSSAs, of course, is that they are specialized for a particular domain and so are only of value if one exists for the domain wherein the engineer is charged with building a new application. While this may make it sound as though DSSAs are rare, the contrary is true. Companies tend to develop multiple applications within a particular area of expertise. This expertise manifests itself in strong similarities in structure between two products from the company and another. These similarities, and the rationale for them, is the basis of a DSSA. What is unfortunately true of such situations, however, is that frequently the common structure—the DSSA—is never written down, but remains latent in the minds of the company's engineers.

DSSAs and product families are explored in Chapter 15 in substantial detail, thus further discussion of these concepts is deferred. The reader should remember, however, that DSSAs are the preeminent means for maximal reuse of knowledge and prior development and are thus critical for developing a new product's design in an established domain.

4.3.2 Architectural Patterns

Architectural patterns are akin to DSSAs, but applied within a much narrower scope. They represent significant reuse of experience, but help you less fully. We repeat here the definition from Chapter 3:

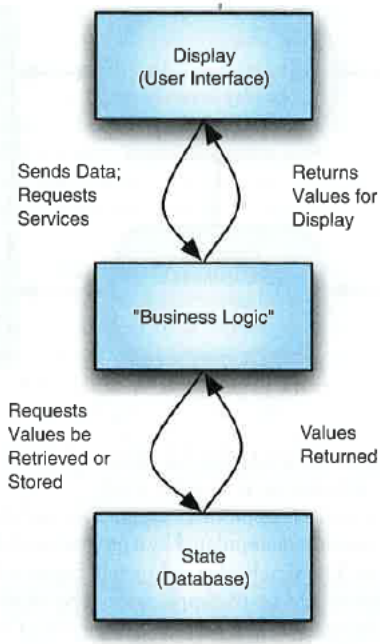
> ***Definition.*** **An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears.**

In the sections below, we sketch three very simple examples: state-logic-display, model-view-controller, and sense-compute-control.

State-Logic-Display (aka. Three-Tier)

The example used in Chapter 3 to illustrate this definition is the familiar three-tier architecture, which is commonly employed in business applications where there is a data store behind a set of business logic rules, and that business logic is accessed by user interface components. This pattern, also known as state-logic-display, has been popular since it maps nicely to a distributed implementation in which communication between the components

Sony Exhibit 1026 Page 19
Sony v. Quantum IPR2023-00954



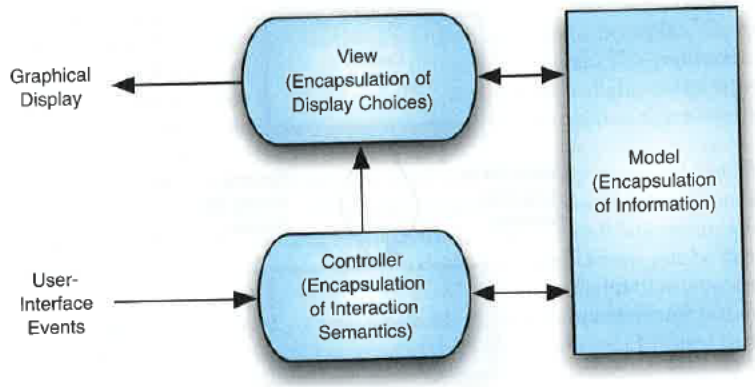
is by remote procedure call. Figure 4-3 shows the pattern, redrawn in an orientation different from that used in Chapter 3.

In business applications, the data store is usually a large database server; the business logic component may implement, for instance, some banking transaction rules, and the display component be a relatively simple component for managing interaction with a user on a desktop PC. Multiplayer games also map nicely to this architecture, wherein the business logic component implements the game rules and the data store maintains the game state. Multiple players can simultaneously play the game wherein each user has his or her own display component. Many Web-based applications can be approximately characterized as three-tier architectures, where the display component is the user's Web browser, the business logic component is the software on the Web site (Web server plus application-specific logic), and the data store is a database accessed by the Web server on a local host. This characterization of Web applications is not entirely fair, however, as communication between the display component (browser) and the business logic component (Web server) is not by means of a remote procedure call, but rather by the HTTP protocol, which does not necessarily maintain a connection between the two components in between interactions.

Model-View-Controller (for Graphical User Interfaces)

Since its invention in the 1980s, the model-view-controller (MVC) pattern has been a dominant influence in the design of graphical user interfaces. While MVC certainly is applied at the low levels of program design and thus could be classified in the design

Sony Exhibit 1026 Page 20
Sony v. Quantum IPR2023-00954



patterns category (and hence omitted from our discussion here), it can also be viewed as providing a structure for applications at a higher level.

The objective of the pattern is to promote separation, and thus independent development paths, between information manipulated by a program and depictions of user interactions with that information. The simple idea is shown in Figure 4-4. The model component encapsulates the information used by the application; the view component encapsulates the information chosen and necessary for graphical depiction of that information; the controller component encapsulates the logic necessary to maintain consistency between the model and the view, and to handle inputs from the user as they relate to the depiction.

The notional interactions between these components are as follows (variations exist in the practice). When the application changes a value in the model object, notification of that change is sent to the view so that any affected parts of the depiction can be updated and redrawn. Notification also typically goes to the controller as well, so that the controller can modify the view if its logic so requires. The view may query the model for additional data needed for the display. When handling input from the user (such as a mouse click on part of the view), the windowing system sends the user event to the controller; the controller may query the view for information to assist in determining what action to take. The controller then updates the model object in keeping with the desired semantics. Then, of course, if the model object changes values it must notify the view and controller so that the user interface can be updated, and so the cycle of interactions continues.

While Figure 4-4 shows three distinct components, from the description above of their interactions it is clear that there is often a very close coupling between the actions of the view component and the controller component. In many cases, therefore, the view and controller components are merged; such is the case with the architecture of Java's Swing framework.

At a much higher level of abstraction than programming, the MVC paradigm can be seen at work in the World Wide Web. Web resources correspond to the model objects; the HTML rendering agent within a browser corresponds to the viewer; the controller in the simplest case corresponds to the code that is part of the browser that responds to user input and which causes either interactions with a Web server or modifies the browser's display in

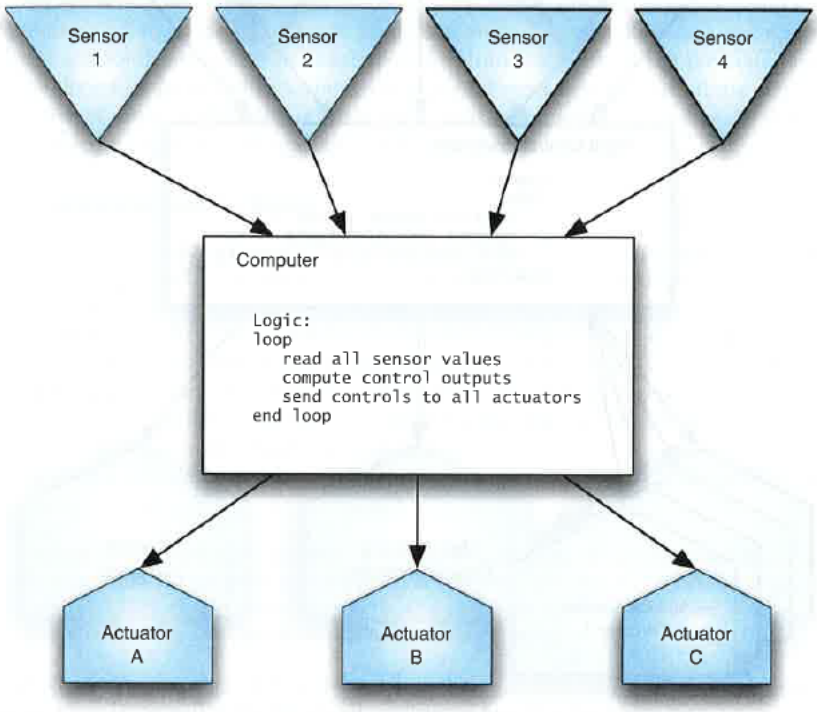
Sony Exhibit 1026 Page 21
Sony v. Quantum IPR2023-00954

some manner. In the more complicated situation, the controller would also encompass code uploaded from a server that governs user interaction with the display, such as JavaScript uploaded as part of a Web page or code on the server that determines the representation of the resource to transmit to the browser.

MVC has been the subject of much work, and a variety of references are available discussing the details of several different implementation approaches. MVC has also inspired the development of other related architectural patterns, such as presentation-abstraction-controller (PAC). Detailed citations are provided at the end of the chapter.

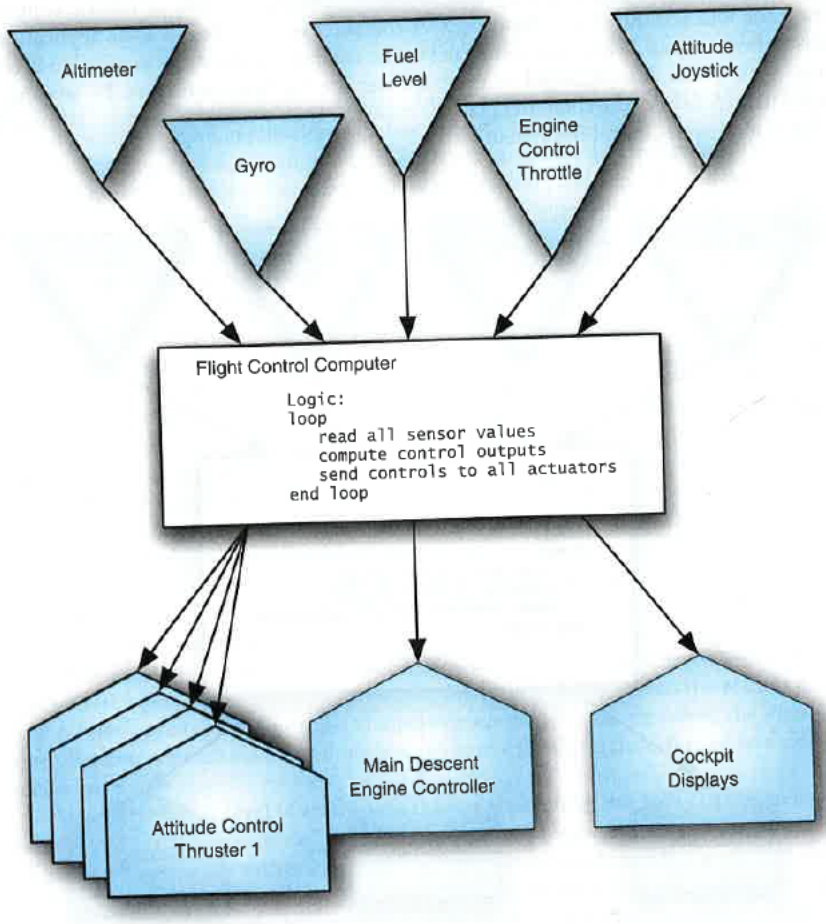
Sense-Compute-Control (aka. Sensor-Controller-Actuator)

The sense-compute-control (SCC) pattern is typically used in structuring embedded control applications. This may range from simple devices such as those found in kitchen appliances to sophisticated systems used in automotive applications or robotic control. The basic idea is illustrated in Figure 4-5. A computer is embedded in some application; sensors from various devices are connected to the computer and may be sampled to determine their value. Also attached to the computer are hardware actuators. By sending a signal to such devices the computer may, for example, open a value, lower flaps, or turn off a light, and hence control the system.



of cycling through the steps of reading all the sensor values, executing a set of control laws or functions, and then sending outputs to the various actuators. Typically, such a cycle is keyed to a clock, where the frequency of the clock ticks may be keyed to the maximum rate at which the sensors' values may change or to the sensitivity of the actuators to varying inputs on their input control lines. *Note that there is implicit feedback in such applications via the external environment.* That is, the typical situation is that by changing one of the actuators, something will change in the external environment that one of the sensors will ultimately detect, and hence cause a change in its value.

simple flight control application can serve to make these ideas concrete. We will use ample of software controlling a notional lunar excursion module, or "lunar lander," surface of the moon, illustrated in Figu



In this example there are five sensors on board the spacecraft: The altimeter (presumably radar) senses the altitude above the surface of the moon; the gyro provides information on the x-y-z axis orientation of the craft; the fuel level indicates the remaining fuel on board; the engine control throttle indicates to the computer the pilot's desired setting of the descent engine's thrust (for instance, 75 percent throttle); and the joystick indicates the pilot's inputs for orienting the spacecraft. (Keep in mind that this is an entirely fly-by-wire application, so there is no physical connection between the joystick, for instance, and any control surfaces. The computer is responsible for sensing all control device inputs.) The diagram shows six actuators: four attitude control thrusters, the main descent engine controller, and the cockpit displays.

As the craft approaches the surface, the sensors will provide to the computer information on the Lunar Lander's altitude, attitude, fuel remaining, and the pilot's control inputs. The flight control computer will process its control laws, computing values to be sent to the various actuators to control the descent engine, the attitude control thrusters, and update the cockpit displays. From changes in the altitude the computer could determine, for example, the descent rate, and display that to the pilot. Based upon the pilot's judgment, he or she could increase the throttle to slow the descent or decrease the throttle and allow the Lunar gravity to increase the descent rate.

Depending on the system designer's choices, a variety of control strategies could be programmed into the flight control computer. The scenario sketched above reflects the strategy applied in the Apollo landings. A fully automated landing sequence could also be programmed, in which the software would determine the optimal descent engine and attitude control thruster settings, based upon the altitude and attitude of the craft. From the standpoint of the software's architecture, these different strategies are all the same; at each cycle of the loop the software reads the current sensor values, computes the desired settings for the actuators, and sends those values to the actuators.

4.3.3 Introduction to Styles

Architectural styles are a primary way of characterizing lessons from experience in software system design. As such, they can be a key element in developing initial or detailed conceptions of a software system's architecture. As Figure 4-2 indicates, architectural styles are broadly applicable; they can be useful in determining everything from subroutine structure to top-level application structure. Styles, as used here and as seen in contrast with the architectural patterns of the preceding section, reflect less domain knowledge than architectural patterns, and hence are more broadly applicable. Keep in mind, however, that the boundaries between these concepts are not precise.

Chapter 3 introduced architectural styles in some detail, and presented the following definition:

> ***Definition.*** **An architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.**

Accordingly, the discussion of styles will include the decisions and constraints comprising the style, as well as the beneficial qualities induced by these choices.

Dewayne Perry and Alexander Wolf wrote early and persuasively on the importance of styles in software architecture. The influence of their work—widely cited in the literature—motivates inclusion of the following extended quote.

> The notion of architectural style is particularly useful from both descriptive and prescriptive points of view. Descriptively, architectural style defines a particular codification of design elements and formal arrangements. Prescriptively, style limits the kinds of design elements and their formal arrangements. That is, an architectural style constrains both the design elements and the formal relationships among the design elements. Analogously, we shall find this a most useful concept in software architecture.
>
> Of extreme importance is the relationship between engineering principles and architectural style (and, of course, architecture itself). For example, one does not get the light, airy feel of the perpendicular style as exemplified in the chapel at King's College, Cambridge, from Romanesque engineering. Different engineering principles are needed to move from the massiveness of the Romanesque to lightness of the perpendicular. It is not just a matter of aesthetics. This relationship between engineering principles and software architecture is also of fundamental importance.
>
> Finally, the relationship between architectural style and materials is of critical importance. The materials have certain properties that are exploited in providing a particular style. One may combine structural with aesthetic uses of materials, such as that found in the post and beam construction of Tudor-style houses. However, one does not build a skyscraper with wooden posts and beams. The material aspects of the design elements provide both aesthetic and engineering bases for an architecture. Again, this relationship is of critical importance in software architecture.
>
> Thus, we find in building architecture some fundamental insights about software architecture: multiple views are needed to emphasize and to understand different aspects of the architecture; styles are a cogent and important form of codification that can be used both descriptively and prescriptively; and, engineering principles and material properties are of fundamental importance in the development and support of a particular architecture and architectural style. . . .
>
> The important thing about an architectural style is that it encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. The useful thing about style is that we can use it both to constrain the architecture and to coordinate cooperating architects. Moreover, style embodies those decisions that suffer erosion and drift. An emphasis on style as a constraint on the architecture provides a visibility to certain aspects of the architecture so that violations of those aspects and insensitivity to them will be more obvious.
>
> (Perry and Wolf 1992)

The architectural styles chosen for presentation here were selected on the basis of reflecting a diversity of approaches, common use, and being a part of the common vocabulary of software architects. Sufficient detail is presented so that the reader can appreciate the essence of each style and the situations for which it is appropriate. While this chapter does not present a comprehensive catalog of styles, many of the most popular and well known are discussed, showing the designer the kind of role that styles can play in solving design problems. (Chapter 11 will explore a few more advanced styles, such as REST, in some detail.)

Sony Exhibit 1026 Page 25
Sony v. Quantum IPR2023-00954