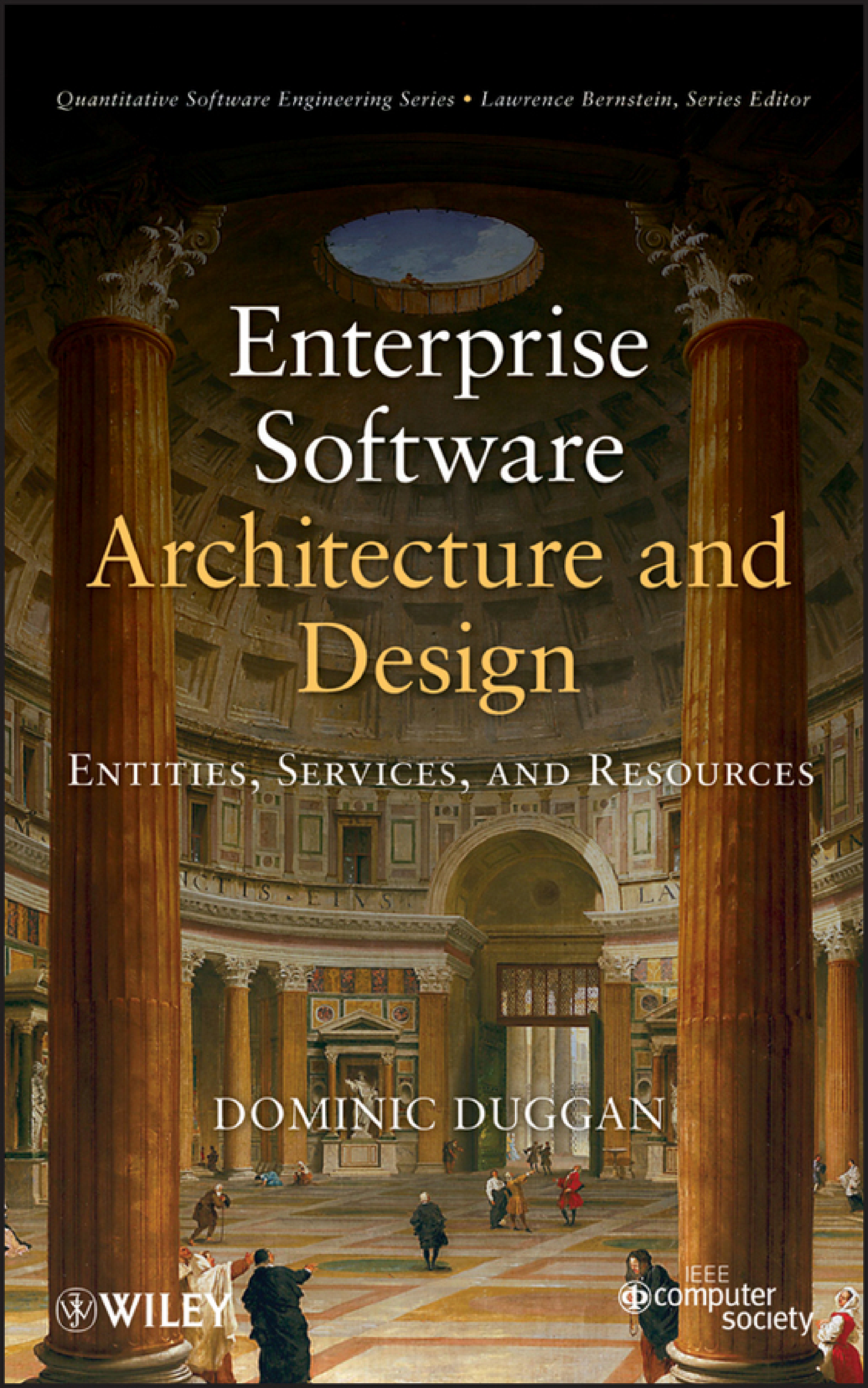
Quantitative Software Engineering Series • Lawrence Bernstein, Series Editor



Enterprise Software Architecture and Design

ENTITIES, SERVICES, AND RESOURCES

DOMINIC DUGGAN

WILEY

IEEE computer society

Enterprise Software Architecture and Design

Quantitative Software Engineering Series

The Quantitative Engineering Series focuses on the convergence of systems engineering with emphasis on quantitative engineering trade-off analysis. Each title brings the principles and theory of programming in-the-large and industrial strength software into focus.

This practical series helps software developers, software engineers, systems engineers, and graduate students understand and benefit from this convergence through the unique weaving of software engineering case histories, quantitative analysis, and technology into the project effort. You will find each publication reinforces the series goal of assisting the reader with producing useful, well-engineered software systems.

Series Editor: **Lawrence Bernstein**

Professor Bernstein is currently an Industry Research Professor at the Stevens Institute of Technology. He previously pursued a distinguished executive career at Bell Laboratories. He is a fellow of the IEEE and ACM.

Trustworthy Systems for Quantitative Software Engineering / Larry Bernstein and C.M. Yuhas

Software Measurement and Estimation: A Practical Approach / Linda M. Laird and M. Carol Brennan

World Wide Web Application Engineering and Implementation / Steven A. Gabarro

Software Performance and Scalability / Henry H. Liu

Managing the Development of Software-Intensive Systems / James McDonald

Trustworthy Compilers / Vladimir O. Safonov

Oracle Database Performance and Scalability: A Quantitative Approach / Henry H. Liu

Enterprise Software Architecture and Design: Entities, Services and Resources / Dominic Duggan

Enterprise Software Architecture and Design

Entities, Services, and Resources

Dominic Duggan

IEEE COMPUTER SOCIETY

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2012 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600, or on the web at 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, (201) 748-6011, fax (201) 748-6008.

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

For general information on our other products and services please contact our Customer Care Department within the U.S. at 877-762-2974 or, outside the U.S. at 317-572-3993 or fax 317-572-4002.

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

Library of Congress Cataloging-in-Publication Data:

Duggan, Dominic.
Enterprises software architecture and design: entities, services, and resources applications / Dominic Duggan.
p. cm.—(Quantitative software engineering series; 10)
ISBN 978-0-470-56545-2 (hardback)
1. Enterprise Software Architecture and Design (Computer science) I. Title.
TK5105.5828.D84 2012
004.6′54—dc23

2011031403

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To My Father

Contents in Brief

1. Introduction 1

2. Middleware 7

3. Data Modeling 59

4. Data Processing 104

5. Domain-Driven Architecture 167

6. Service-Oriented Architecture 207

7. Resource-Oriented Architecture 359

Appendix A: Introduction to Haskell 416

Appendix B: Time in Distributed Systems 437

Contents

List of Figures **xv**

Acknowledgements **xxiii**

1. Introduction **1**

References / 6

2. Middleware **7**

2.1 Enterprise Information Systems / 7

2.2 Communication / 12

2.3 System and Failure Models / 21

2.4 Remote Procedure Call / 34

2.5 Message-Oriented Middleware / 42

2.6 Web Services and Service-Oriented Architecture (SOA) / 46

2.7 Cloud Computing / 52

2.8 Naming and Discovery / 55

2.9 Further Reading / 56

References / 57

3. Data Modeling **59**

3.1 Entities and Relationships / 60
3.1.1 Concepts and Entities / 60
3.1.2 Attributes and Relationships / 61
3.1.3 Properties of Relationship Types / 65
3.1.4 Special Relationship Types / 69

3.2 XML Schemas / 74

3.3 Defining New Types / 79
3.3.1 Defining Simple Types / 79
3.3.2 Defining Complex Types / 82

3.4 Derived Types / 85
3.4.1 Derived Simple Types / 86
3.4.2 Derived Complex Types / 87

3.5 Document Hierarchies / 94

3.6 Relationship Types in XML Schemas / 98

3.7 Metaschemas and Metamodels / 100

3.8 Further Reading / 102

References / 102

4. Data Processing **104**

4.1 Processing XML Data / 104
4.1.1 Tree Processing / 105
4.1.2 Schema Binding / 109
4.1.3 Stream Processing / 114
4.1.4 External Processing / 119

4.2 Query Languages and XQuery / 122

4.3 XML Databases / 134
4.3.1 Storage as Relational Tables / 135
4.3.2 Storage as Large Strings / 137
4.3.3 Native XML Storage / 137

4.4 Web Services / 138
4.4.1 SOAP: (not so) Simple Object Access Protocol / 139
4.4.2 WSDL: Web Services Description Language / 145
4.4.3 Web Service Policy / 155

4.5 Presentation Layer: JSON and JQUERY / 159

References / 166

5. Domain-Driven Architecture **167**

5.1 Software Architecture / 167

5.2 Domain-Driven Design / 168

5.3 Application Frameworks / 175

5.4 Domain-Specific Languages (DSLs) / 180

5.5 An Example API for Persistent Domain Objects / 188

5.6 Domain-Driven Architecture / 197

5.7 Further Reading / 205

References / 205

6. Service-Oriented Architecture **207**

6.1 Services and Procedures / 207

6.2 Service-Oriented Architecture (SOA) / 211

6.3 Service Design Principles / 216

6.4 Service-Oriented Architecture (SOA) Governance / 218

6.5 Standardized Service Contract / 221

6.5.1 Operations Contract / 222

6.5.2 Data Contract / 223

6.5.3 Policy Contract / 224

6.5.4 Binding Contract / 226

6.5.5 Contract Versioning / 231

6.6 Service Loose Coupling / 237

6.6.1 Motivation for Loose Coupling / 237

6.6.2 Contract Development / 239

6.6.3 Loose Coupling Patterns / 242

6.6.4 Cost of Loose Coupling / 246

6.7 Service Abstraction / 248

6.7.1 Platform Abstraction / 248

6.7.2 Protocol Abstraction / 249

6.7.3 Procedural Abstraction / 261

6.7.4 State Abstraction / 264
6.7.5 Data Abstraction / 269
6.7.6 Endpoint Abstraction / 278

6.8 Service Reusability / 278
6.8.1 Parameterization and Bounded Polymorphism / 279
6.8.2 Subtyping, Inheritance, and Contracts / 284
6.8.3 Does Service-Oriented Architecture Require Subtyping? / 289
6.8.4 Patterns for Service Reusability / 292

6.9 Service Autonomy / 299
6.9.1 Replicating Computation / 300
6.9.2 Replicating State / 303
6.9.3 Sources of Errors and Rejuvenation / 308
6.9.4 Caching / 313

6.10 Service Statelessness / 323
6.10.1 Contexts and Dependency Injection / 331

6.11 Service Discoverability / 336
6.11.1 Global Discovery / 336
6.11.2 Local Discovery / 337
6.11.3 Layered Naming / 347

6.12 Further Patterns / 351

6.13 Further Reading / 352

References / 352

7. Resource-Oriented Architecture 359

7.1 Representational State Transfer / 359

7.2 RESTful Web Services / 369

7.3 Resource-Oriented Architecture (ROA) / 379

7.4 Interface Description Languages / 387
7.4.1 Web Services Description Language (WSDL) / 387
7.4.2 Web Application Description Language (WADL) / 390

7.5 An Example Application Program Interface (API) for Resource-Oriented Web Services / 396

7.6 Hypermedia Control and Contract Conformance / 406

7.7 Concluding Remarks / 412

7.8 Further Reading / 414

References / 414

Appendix A: Introduction to Haskell **416**

A.1 Types and Functions / 416

A.2 Type Classes and Functors / 425

A.3 Monads / 431

A.4 Further Reading / 436

References / 436

Appendix B: Time in Distributed Systems **437**

B.1 What Time Is It? / 437

B.2 Time and Causality / 443

B.3 Applications of Logical and Vector Time / 450

B.3.1 Mutual Exclusion / 450

B.3.2 Quorum Consensus / 451

B.3.3 Distributed Logging / 456

B.3.4 Causal Message Delivery / 458

B.3.5 Distributed Snapshots / 463

B.4 Virtual Time / 468

B.5 Further Reading / 470

References / 470

Index **473**

List of Figures

2.1 Single-tier systems 8

2.2 Two-tier (client-server) systems 9

2.3 Three-tier systems 11

2.4 Enterprise network 12

2.5 End-to-end versus point-to-point error correction 13

2.6 Open Systems Interconnection (OSI) network protocol stacks and Web protocols 14

2.7 Message-passing semantics 17

2.8 Sliding window protocols for pipeline parallelism 19

2.9 System models 22

2.10 Architecture for a load-balancing Web service 23

2.11 Failure detectors 25

2.12 Consensus with unreliable failure detector 29

2.13 Failure detectors and distributed problems 30

2.14 Byzantine Agreement protocol 33

2.15 Compiling and linking client stubs and server skeletons 35

2.16 Invoking services via stubs 36

2.17	Compound RPC	37
2.18	Service discovery	37
2.19	Failure scenarios with RPC	39
2.20	Limitations of using acknowledgements for coordination	40
2.21	Message brokers	45
2.22	Service-oriented architecture	50
2.23	SOA via Web services and enterprise service bus	51
3.1	Concepts and models	60
3.2	Relationships versus attributes in UML	62
3.3	Storing XML data in relational form	64
3.4	One-to-many and many-to-many relationships	65
3.5	*n*-ary relationships	66
3.6	*n*-ary relationships	68
3.7	Reification with association classes	70
3.8	Aggregation	71
3.9	Composition	71
3.10	Generalization	72
3.11	Specialization	73
3.12	XML schemas with element declarations and type definitions	76
3.13	Instance document	79
3.14	Examples of datatype definitions	80
3.15	Examples of compound datatype definitions	81
3.16	Schema with choice element	82
3.17	Instance document for schema in Fig. 3.16	83
3.18	Declaration of attributes	84
3.19	Inclusion among simple subtypes	87
3.20	Derived complex types	89
3.21	Example instance for schema of Fig. 3.20	90
3.22	Inclusion among complex subtypes	90

3.23 Element substitution 92
3.24 Type substitutability and element substitution 93
3.25 Derived complex type based on derived simple content type 93
3.26 Derivation hierarchy for Fig. 3.25 94
3.27 Example of including schemas 95
3.28 Example instance for schema in Fig. 3.27 96
3.29 Example schemas to be imported 96
3.30 Importing schemas 97
3.31 Example instance for schema in Fig. 3.30 (a) with `elementFormDefault="unqualified"` 98
3.32 Representing one-to-many using embedding 99
3.33 Representing many-to-many relationship types using keys 100
3.34 Schemas and meta-schemas 101
3.35 Meta-entities for media types 102
4.1 Processing XML data 106
4.2 Document object model 107
4.3 Catalog schema 110
4.4 JAXB mapping for schema 111
4.5 Alternative schema representation 114
4.6 JIBX mapping rules 115
4.7 Boxed and unboxed representations with JIBX 116
4.8 SAX architecture 116
4.9 XSLT example 120
4.10 XSLT in enterprise architecture 121
4.11 Catalog example 124
4.12 Review schema 129
4.13 Review database 130
4.14 Synthesis of documents using XQuery 134
4.15 In-memory native XML database in application server 135
4.16 Interactions in SOAP-based Web services 139

4.17 Web services stack 140
4.18 Structure of a SOAP message 140
4.19 SOAP request 141
4.20 SOAP response 143
4.21 SOAP fault 144
4.22 Components of a WSDL interface description 146
4.23 Message exchanges with reviews service 147
4.24 Reviews service: abstract part 148
4.25 Reviews service: concrete part 151
4.26 Reviews service: specification for posting callback 152
4.27 Review posting service: abstract part 153
4.28 JSON and JTD example 160
4.29 Representing XML data in JSON 164
4.30 Examples of jQuery selectors 165
5.1 Repository pattern 172
5.2 Anti-corruption layer 173
5.3 Context map 174
5.4 Instantiating a framework 179
5.5 Interfaces for financial instruments 180
5.6 Risla loan program [11] 181
5.7 Primitives for Haskell contracts [12] 183
5.8 Patient entities and relationships 189
5.9 Entity class for patient 190
5.10 Auxiliary class declarations 192
5.11 API for Java persistence architecture 193
5.12 Data access object 194
5.13 Robustness diagram 198
5.14 Domain-driven architecture 199
5.15 Domain-driven architecture with proxy PDOs 201

5.16	Domain-driven architecture with detached PDOs	202
5.17	Domain-driven architecture with fat client	204
6.1	Constructors and observers for lists	209
6.2	Constructor-oriented representation of lists	210
6.3	Observer-oriented representation of lists	211
6.4	Service-oriented architecture	212
6.5	Auction services	214
6.6	Business logic service	215
6.7	Data logic service	215
6.8	Service and inventory design patterns	218
6.9	Policy enforcement for SOA governance [14]	221
6.10	Binding contracts for an Enterprise Java Bean (EJB)	229
6.11	Internal and external communications in a server cluster	230
6.12	Zones, change absorbers, and a round-trip message	235
6.13	Service contract development	240
6.14	WSDL specification in an XML editor	240
6.15	Customer profiling services	243
6.16	Connector architecture for legacy integration	244
6.17	Wrapping legacy system as a service	245
6.18	Example interactions in an RPC stack	247
6.19	WCF service definition	251
6.20	WCF service definition with asynchronous method implementation	253
6.21	WCF service definition with duplex method	254
6.22	Asynchronous shopping cart operations in Java EE	256
6.23	Asynchronous shopping cart operations in WebLogic	258
6.24	Exception handling	262
6.25	Region and effect based memory management in Cyclone	264
6.26	State abstraction in constructor-oriented versus observer-oriented paradigms	265

6.27 Purchase order with typestate interface 266
6.28 Data abstraction in constructor-oriented versus observer-oriented paradigms 270
6.29 Abstract data types via opaque types in O'CAML 274
6.30 Partially abstract types in Modula-3 276
6.31 Abstract data types via sealing in Not-O'CAML 277
6.32 Implementation reuse via parameterized modules in O'CAML 280
6.33 Implementation and inheritance relationships 285
6.34 Subtyping of service operation types 287
6.35 Uses of subtyping 291
6.36 Multi-channel health portal service 296
6.37 Certificate authority service layer 297
6.38 Redundant implementation: physical and virtual 301
6.39 Quorum consensus 305
6.40 Ordered update 306
6.41 State transitions with and without rejuvenation 310
6.42 Failure to commit client updates on an NFS server 315
6.43 Read session semantics 318
6.44 Write session semantics 319
6.45 Combining Read Your Writes (RYW) and Writes Follow Reads (WFR) 320
6.46 Commuicating updates 322
6.47 Collaborative editing based on operation transformations 323
6.48 Sources of state deferral in a cluster 327
6.49 Stateful services in WCF 328
6.50 Stateful services in Java EE 329
6.51 Ad hoc discovery in WCF 344
6.52 WCF managed discovery 346
6.53 Naming and discovery 349
6.54 Naming and discovery 351

7.1 SOAP versus REST Web services 363
7.2 Flickr API 371
7.3 Train reservation service: type definitions 388
7.4 Train reservation interface: interface and bindings 389
7.5 WADL example: resources 391
7.6 WADL example: get train schedule 392
7.7 WADL example: make train reservation 393
7.8 Train schedule resource 397
7.9 Train schedule service 398
7.10 Train schedule representation 399
7.11 Train and reservation resources 401
7.12 Stateful shopping cart 402
7.13 RESTful shopping cart 403
7.14 Server-side hypermedia controller 407
7.15 Client-side hypermedia controller interface 408
B.1 PTP spanning tree 441
B.2 Path asymmetry in polling time server 442
B.3 Example timeline 445
B.4 Logical clocks protocol 446
B.5 Logical time example 447
B.6 Vector clocks protocol 449
B.7 Vector time example 450
B.8 Ordered updates based on Voting Protocol 452
B.9 Ordered updates based on Viewstamp Protocol 454
B.10 Logging distributed events for debugging 457
B.11 Causal message delivery 459
B.12 Causal unicast and multicast 460
B.13 Causal and total multicast 461
B.14 FIFO and causal order 462

B.15 Problems with snapshots 463
B.16 Snapshot protocol 464
B.17 State transitions and snapshot state 467
B.18 Virtual time 468

Acknowledgements

Thanks to Larry Bernstein for suggesting this book, and for his friendship.
Thanks to my students for their enthusiasm.
Thanks to my family for their support and encouragement.

D. D.

1

Introduction

Communications protocol development has tended to follow two paths. One path has emphasized integration with the current language model. The other path has emphasized solving the problems inherent in distributed computing. Both are necessary, and successful advances in distributed computing synthesize elements from both camps.
Waldo et al. [1]

This book is about *programming-in-the-large*, a term coined by DeRemer and Kron [2] to distinguish the assembly of large applications from components, from the task of implementing those components themselves. Many of the principles of programming-in-the-large were earlier elucidated by Parnas in two articles [3, 4]. It is striking how many of the key ideas of programming-in-the-large that remain relevant today were in Parnas' seminal work.

In the 1980s, with the rise of local area networks (LANs) and object-oriented programming, distributed software buses appeared as solutions to the growing complexity of networked heterogeneous software systems. The synthesis of these trends saw the emergence of the Common Object Request Broker Architecture (CORBA) in the late 1980s. Long before then, industries such as banking and air travel had developed on-line networked transactional systems that are still in heavy use today. CORBA, as originally envisaged, was to be the glue for allowing those legacy systems to be incorporated into modern applications. The use of systems such as CORBA, and particularly of transaction processing monitors, as the backbone for enterprise middlewares has, indeed, been a success story in enterprise software environments. Unfortunately, and perhaps predictably, the challenge of heterogeneity remained, as a heterogeneous collection of software buses emerged for application deployment.

Enterprise Software Architecture and Design: Entities, Services, and Resources,
First Edition. Dominic Duggan.
© 2012 John Wiley & Sons, Inc. Published 2012 by John Wiley & Sons, Inc.

In the 1990s, a true sea change happened with the advent of the World Wide Web. Besides making the uniform resource locator (URL) a part of the daily culture, the spectacular success of the Web has required enterprises to consider how their business model can adapt to this newly popular communication medium. The first Netscape browser included support for the hypertext transfer protocol (HTTP) for the Web, but also supported the safe execution of dynamically downloaded Java applets in the Web browser. It also had support for the CORBA communication protocol to enable those applets to connect back to the server through sophisticated inter-object request broker protocols. Clearly, the thinking here was that a simple protocol, such as the Web protocol, should just be an entry point into the more sophisticated realm of distributed enterprise applications built on modern middleware.

The vision of ubiquitous distributed object systems connecting the world together did not materialize. It became impossible to ignore the significance of location that distributed object systems were supposed to mask. No amount of middleware layering can hide the fact that there is a network connecting the points together, and that network has effects such as latency and failures that cannot be masked from applications. At the same time, the Web protocols that were supposed to be just an entry point into more sophisticated protocols showed amazing staying power, while frustration grew with the complexity of the more "sophisticated" approaches. Most importantly, connecting applications across enterprise boundaries remains a difficult problem, perhaps inevitably so because market forces push against a standardized solution to the problem.

The first decade of the 21st century saw the rise of Web services, an alternative approach to distributed object systems for providing and consuming software services outside the enterprise. Rather than run over proprietary protocol stacks, Web services were proposed to operate over the same Web protocol that had been such a spectacular success for business-to-consumer e-commerce. Although business-to-business e-commerce over reliable messaging systems has been in use for many years, the clear trend has been to vastly increase the scale of this inter-operation by making use of the Web protocol stack. Enterprises must learn to leverage the Internet for rapid and agile application development by continually striving for the optimum mix of in-house expertise and outsourced software services. Thus, service-oriented architecture (SOA) emerged: partly driven by the enabling technology of Web services, partly building on work in component-based development for programming-in-the-large, and partly seeking to align the information technology (IT) architecture with the business goals of the enterprise.

The experience of the WS-* standardization effort has been an interesting one. Developers in the field resisted adoption of the WS-* stack, essentially citing the end-to-end argument underlying the original design of the Internet to decry the complexity of the various layers in the stack. The controversy has had an evident effect. Despite early enthusiasm for simple object access protocol (SOAP)- and Web services description language (WSDL)-based Web services, many companies have pulled back and are instead adopting so-called RESTful (representational state transfer) Web services. As an indication of the "zeitgeist", one IT writer went so far as to declare the SOAP protocol stack to be the worst technology of the last decade [5].

Yet, despite the fact that difficult challenges remain, there is room for optimism. For all of its flaws, asynchronous JavaScript and XML (AJAX) has clearly moved the ball forward in terms of building responsive networked applications where code is easily deployed to where it needs to be executed—essentially anywhere on the planet. As we shall see in Chapter 5, this trend looks likely to intensify. Meanwhile, tools such as jQuery have emerged to tame the complexities of clumsier tools such as the Document Object Model (DOM), while RESTful Web services have emerged as a backlash against the complexity of SOAP-based Web services. If there is a lesson here, it may, perhaps, be in E. F. Schumacher's famous admonition: "Small is beautiful" [6]. Just because a multi-billion dollar industry is pushing a technology, does not necessarily mean developers will be forced to use it, and there is room for the individual with key insight and the transforming idea to make a difference.

However, platforms such as the SOAP stack have at least had a real motivation. Architects and developers tend to have opposing ideas on the subject of SOAP versus REST. REST, or some derivative of REST, may be argued to be the best approach for building applications over the internet, where challenges such as latency and independent failures are unavoidable. Nevertheless, the issues of enterprise collaboration and inter-operation that motivated SOAP and WSDL, however imperfect those solutions were, are not going away. Indeed, the move to cloud computing, which increasingly means outsourcing IT to third parties, ensures that these issues will intensify in importance for enterprises. The criticality of IT in the lives of consumers will also intensify as aging populations rely on healthcare systems that make increasing use of IT for efficiency, cost savings and personalized healthcare. Similarly, IT will play a crucial role in solving challenges with population pressure and resource usage that challenge all of mankind. For example, Bill Gates has made the case for improved healthcare outcomes as critical to reducing family size in developing countries; IT is a key component in delivering on these improved outcomes.

The intention of this book is to cover the principles underlying the enterprise software systems that are going to play an increasing part in the lives of organizations of all kinds and of individuals. Some of the issues that SOA claims to address have been with us for decades. It is instructive, therefore, while discussing the problems and proposed solutions, to also ensure that there is agreement on the definition of the principles. However the book must also be relevant and discuss current approaches. Part of the challenge is that, following the battles of the last decade, it is not clear what role technologies such as Java Enterprise Edition (Java EE) and Windows Communication Foundation (WCF), so central to the adoption of SOA, will play in future enterprise systems. Therefore, this book emphasizes principles in a discussion of some of the current practices, with a critical eye towards how the current approaches succeed or fail to live up to those principles, with passing speculation about how matters may be improved.

In this text the focus is on data modeling and software architecture, particularly SOA. SOA clearly dominates the discussion. This is not intended to denigrate any of the alternative architectural styles, but SOA brings many traditional issues in software engineering to the forefront in enterprise software architecture, and it has ambitious goals for aligning IT strategy with business goals. Domain-driven design is essentially object-oriented design, and it is principally discussed in order to counterpoint it with SOA. There is reason to believe that domain-driven architecture will play a more prominent role in enterprise software architecture, as experience with mobile code progresses. Resource-oriented architecture (ROA) is certainly the technology of choice for many developers, but it is not clear how well it will address the needs of application architects attempting to deal with the issues in enterprise applications. Many of the issues in software architecture highlighted by SOA will also eventually need to be addressed by ROA.

As noted by Waldo et al., the development of enterprise applications has broadly followed two courses: the "systems" approach, in which developers have focused on the algorithmic problems to be solved in building distributed systems, and the "programming environment" approach, which has attempted to develop suitable tools for enterprise application development. The success of the former has been substantially greater than the latter, where the retreat from SOAP-based to REST-based Web services can be viewed as the most recent failure to develop an acceptable programming model. Much of the focus of the systems approach has been on the proper algorithms and protocols for providing reliability and, increasingly, security for enterprise applications. The use of these tools in application development is still a challenge. Computer scientists, sometimes express surprise when they learn of the importance of these tools in the infrastructure of the cloud.

Service developers wrestle with the problems of network security and independent failures every day—even if the solutions are sometimes flawed because of imperfect understanding of the problems and solutions. While the last several decades have seen attempts to build middleware platforms that encapsulate the use of these tools, isolating application developers from the problems with distributed applications, the REST philosophy challenges the assumptions underlying this approach, at least for applications that run over the Web, based on an end-to-end argument that focuses responsibility for dealing with the issues on the application rather than the system. This is a basic rejection of three decades of distributed programming platforms, and challenges the research and developer community to develop the appropriate abstractions that make this effort scalable. Certainly, the application developer must have the appropriate tools provided in programming frameworks. However, even with these tools, the developer still has the responsibility of using and deploying these tools in their application. This approach is in opposition to the traditional middleware approach, which purports to allow developers to build applications without regard to distribution and rely on the middleware to handle the distribution aspects.

The material in this text comes primarily from two teaching courses: first, and primarily, a course on software architecture for enterprise applications; secondly,