SOA
Principles of Service Design

Thomas Erl

PRENTICE HALL
UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO
NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID
CAPETOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

Contents

Preface xxv

Chapter 1: Introduction 1

1.1 Objectives of this Book 3

1.2 Who this Book Is For 3

1.3 What this Book Does Not Cover 4

Topics Covered by Other Books 4

SOA Standardization Efforts 5

1.4 How this Book Is Organized 6

Part I: Fundamentals 7

Part II: Design Principles 9

Part III: Supplemental 12

Appendices 12

1.5 Symbols, Figures, and Style Conventions 13

Symbol Legend 13

How Color Is Used 13

The Service Symbol 13

1.6 Additional Information 16

Updates, Errata, and Resources (www.soabooks.com) 16

Master Glossary (www.soaglossary.com) 16

Referenced Specifications (www.soaspecs.com) 16

Service-Oriented Computing Poster (www.soaposters.com) 16

The SOA Magazine (www.soamag.com) 17
Notification Service 17
Contact the Author 17

Chapter 2: Case Study 19

2.1 Case Study Background: Cutit Saws Ltd. 20
History 20
Technical Infrastructure and Automation Environment 21
Business Goals and Obstacles 21

PART I: FUNDAMENTALS

Chapter 3: Service-Oriented Computing and SOA. 25

3.1 Design Fundamentals 26
Design Characteristic 27
Design Principle 28
Design Paradigm 29
Design Pattern. 30
Design Pattern Language 31
Design Standard 32
Best Practice 34
A Fundamental Design Framework. 35

3.2 Introduction to Service-Oriented Computing 37
Service-Oriented Architecture. 38
Service-Orientation, Services, and Service-Oriented Solution Logic 39
Service Compositions 39
Service Inventory. 40
Understanding Service-Oriented Computing Elements 40
Service Models 43
SOA and Web Services. 46
Service Inventory Blueprints 51
Service-Oriented Analysis and Service Modeling. 52

Service-Oriented Design . . . 53
Service-Oriented Architecture: Concepts, Technology, and Design . . . 54
3.3 Goals and Benefits of Service-Oriented Computing . . . 55
Increased Intrinsic Interoperability . . . 56
Increased Federation . . . 58
Increased Vendor Diversification Options . . . 59
Increased Business and Technology Domain Alignment . . . 60
Increased ROI . . . 61
Increased Organizational Agility . . . 63
Reduced IT Burden . . . 64
3.4 Case Study Background . . . 66

Chapter 4: Service-Orientation . . . 67

4.1 Introduction to Service-Orientation . . . 68
Services in Business Automation . . . 69
Services Are Collections of Capabilities . . . 69
Service-Orientation as a Design Paradigm . . . 70
Service-Orientation and Interoperability . . . 74
4.2 Problems Solved by Service-Orientation . . . 75
Life Before Service-Orientation . . . 76
The Need for Service-Orientation . . . 81
4.3 Challenges Introduced by Service-Orientation . . . 85
Design Complexity . . . 85
The Need for Design Standards . . . 86
Top-Down Requirements . . . 86
Counter-Agile Service Delivery in Support of Agile Solution Delivery . . . 87
Governance Demands . . . 88
4.4 Additional Considerations . . . 89
It Is Not a Revolutionary Paradigm . . . 89
Enterprise-wide Standardization Is Not Required . . . 89
Reuse Is Not an Absolute Requirement . . . 90

4.5 Effects of Service-Orientation on the Enterprise 91
Service-Orientation and the Concept of "Application" 91
Service-Orientation and the Concept of "Integration" 92
The Service Composition 94
Application, Integration, and Enterprise Architectures 95
4.6 Origins and Influences of Service-Orientation 96
Object-Orientation 97
Web Services 98
Business Process Management (BPM) 98
Enterprise Application Integration (EAI) 98
Aspect-Oriented Programming (AOP) 99
4.7 Case Study Background 100

Chapter 5: Understanding Design Principles 103

5.1 Using Design Principles 104
Incorporate Principles within Service-Oriented Analysis 105
Incorporate Principles within Formal Design Processes 106
Establish Supporting Design Standards 107
Apply Principles to a Feasible Extent 108
5.2 Principle Profiles 109
5.3 Design Pattern References 111
5.4 Principles that Implement vs. Principles that Regulate . . . 111
5.5 Principles and Service Implementation Mediums 114
"Capability" vs. "Operation" vs. "Method" 115
5.6 Principles and Design Granularity 115
Service Granularity 116
Capability Granularity 116
Data Granularity 116
Constraint Granularity 117
Sections on Granularity Levels 118
5.7 Case Study Background 119
The Lab Project Business Process 119

PART II: DESIGN PRINCIPLES

Chapter 6: Service Contracts (Standardization and Design) 125

6.1 Contracts Explained 126
- Technical Contracts in Abstract 126
- Origins of Service Contracts 127

6.2 Profiling this Principle 130

6.3 Types of Service Contract Standardization 132
- Standardization of Functional Service Expression 133
- Standardization of Service Data Representation 134
- Standardization of Service Policies 137

6.4 Contracts and Service Design 140
- Data Representation Standardization and Transformation Avoidance 140
- Standardization and Granularity 142
- Standardized Service Contracts and Service Models 144
- How Standardized Service Contract Design Affects Other Principles 144

6.5 Risks Associated with Service Contract Design 149
- Versioning 149
- Technology Dependencies 150
- Development Tool Deficiencies 151

6.6 More About Service Contracts 152
- Non-Technical Service Contract Documents 152
- "Web Service Contract Design for SOA" 153

6.7 Case Study Example 154
- Planned Services 154
- Design Standards 155
- Standardized WSDL Definition Profiles 155
- Standardized XML Schema Definitions 157
- Standardized Service and Data Representation Layers 157
- Service Descriptions 158
- Conclusion 160

Chapter 7: Service Coupling (Intra-Service and Consumer Dependencies) 163

7.1 Coupling Explained 164
Coupling in Abstract 165
Origins of Software Coupling 165
7.2 Profiling this Principle 167
7.3 Service Contract Coupling Types 169
Logic-to-Contract Coupling (the coupling of service logic to the service contract) 173
Contract-to-Logic Coupling (the coupling of the service contract to its logic) 174
Contract-to-Technology Coupling (the coupling of the service contract to its underlying technology) 176
Contract-to-Implementation Coupling (the coupling of the service contract to its implementation environment) 177
Contract-to-Functional Coupling (the coupling of the service contract to external logic) 180
7.4 Service Consumer Coupling Types 181
Consumer-to-Implementation Coupling 182
Standardized Service Coupling and Contract Centralization . . . 185
Consumer-to-Contract Coupling 185
Measuring Consumer Coupling 191
7.5 Service Loose Coupling and Service Design 193
Coupling and Service-Orientation 193
Service Loose Coupling and Granularity 195
Coupling and Service Models 196
How Service Loose Coupling Affects Other Principles 197
7.6 Risks Associated with Service Loose Coupling 200
Limitations of Logic-to-Contract Coupling 200
Problems when Schema Coupling Is “too loose” 201
7.7 Case Study Example 202
Coupling Levels of Existing Services 202
Introducing the InvLegacyAPI Service 203
Service Design Options 205

Chapter 8: Service Abstraction (Information Hiding and Meta Abstraction Types) 211

8.1 Abstraction Explained 212
Origins of Information Hiding 213
8.2 Profiling this Principle 214
Why Service Abstraction Is Needed 214
8.3 Types of Meta Abstraction 218
Technology Information Abstraction 219
Functional Abstraction 221
Programmatic Logic Abstraction 222
Quality of Service Abstraction 224
Meta Abstraction Types and the Web Service Regions of Influence 225
Meta Abstraction Types in the Real World 227
8.4 Measuring Service Abstraction 231
Contract Content Abstraction Levels 231
Access Control Levels 232
Abstraction Levels and Quality of Service Meta Information . . . 234
8.5 Service Abstraction and Service Design 235
Service Abstraction vs. Service Encapsulation 235
How Encapsulation Can Affect Abstraction 235
Service Abstraction and Non-Technical Contract Documents . . 237
Service Abstraction and Granularity 238
Service Abstraction and Service Models 239
How Service Abstraction Affects Other Principles 239
8.6 Risks Associated with Service Abstraction 242
Multi-Consumer Coupling Requirements 242
Misjudgment by Humans 242
Security and Privacy Concerns 243
8.7 Case Study Example 244
Service Abstraction Levels 244
Operation-Level Abstraction Examples 247

Chapter 9: Service Reusability (Commercial and Agnostic Design) 253

9.1 Reuse Explained 254
Reuse in Abstract 254
Origins of Reuse 257

9.2 Profiling this Principle 259

9.3 Measuring Service Reusability and Applying Commercial Design 262
Commercial Design Considerations 262
Measures of Planned Reuse 265
Measuring Actual Reuse 267
Commercial Design Versus Gold-Plating 267

9.4 Service Reuse in SOA 268
Reuse and the Agnostic Service 268
The Service Inventory Blueprint 269

9.5 Standardized Service Reuse and Logic Centralization . . 270
Understanding Logic Centralization 271
Logic Centralization as an Enterprise Standard 272
Logic Centralization and Contract Centralization 272
Centralization and Web Services 274
Challenges to Achieving Logic Centralization 274

9.6 Service Reusability and Service Design 276
Service Reusability and Service Modeling 276
Service Reusability and Granularity 277
Service Reusability and Service Models 278
How Service Reusability Affects Other Principles 278

9.7 Risks Associated with Service Reusability and Commercial Design 281
Cultural Concerns 281
Governance Concerns 283
Reliability Concerns 286
Security Concerns 286
Commercial Design Requirement Concerns 286
Agile Delivery Concerns 287

9.8 Case Study Example 288
The Inventory Service Profile 288
Assessing Current Capabilities 289
Modeling for a Targeted Measure of Reusability 289
The New EditItemRecord Operation 290
The New ReportStockLevels Operation 290
The New AdjustItemsQuantity Operation 291
Revised Inventory Service Profile 292

Chapter 10: Service Autonomy (Processing Boundaries and Control) 293

10.1 Autonomy Explained 294
Autonomy in Abstract 294
Origins of Autonomy 295
10.2 Profiling this Principle 296
10.3 Types of Service Autonomy 297
Runtime Autonomy (execution) 298
Design-Time Autonomy (governance) 298
10.4 Measuring Service Autonomy 300
Service Contract Autonomy (services with normalized contracts) 301
Shared Autonomy 305
Service Logic Autonomy (partially isolated services) 306
Pure Autonomy (isolated services) 308
Services with Mixed Autonomy 310
10.5 Autonomy and Service Design 311
Service Autonomy and Service Modeling 311
Service Autonomy and Granularity 311
Service Autonomy and Service Models 312
How Service Autonomy Affects Other Principles 314
10.6 Risks Associated with Service Autonomy 317
Misjudging the Service Scope 317
Wrapper Services and Legacy Logic Encapsulation 318
Overestimating Service Demand 318

10.7 Case Study Example 319
Existing Implementation Autonomy of the GetItem Operation . . 319
New Operation-Level Architecture with Increased Autonomy . . 320
Effect on the Run Lab Project Composition 322

Chapter 11: Service Statelessness (State Management Deferral and Stateless Design) 325

11.1 State Management Explained 327
State Management in Abstract 327
Origins of State Management 328
Deferral vs. Delegation 331
11.2 Profiling this Principle 331
11.3 Types of State 335
Active and Passive 335
Stateless and Stateful 336
Session and Context Data 336
11.4 Measuring Service Statelessness 339
Non-Deferred State Management (low-to-no statelessness) . . . 340
Partially Deferred Memory (reduced statefulness) 340
Partial Architectural State Management Deferral (moderate statelessness) 341
Full Architectural State Management Deferral (high statelessness) 342
Internally Deferred State Management (high statelessness) . . . 342
11.5 Statelessness and Service Design 343
Messaging as a State Deferral Option 343
Service Statelessness and Service Instances 344
Service Statelessness and Granularity 346
Service Statelessness and Service Models 346
How Service Statelessness Affects Other Principles 347
11.6 Risks Associated with Service Statelessness 349
Dependency on the Architecture 349
Increased Runtime Performance Demands 350
Underestimating Delivery Effort 350

11.7 Case Study Example 351
Solution Architecture with State Management Deferral 352
Step 1 353
Step 2 354
Step 3 355
Step 4 356
Step 5 357
Step 6 358
Step 7 359

Chapter 12: Service Discoverability (Interpretability and Communication) 361

12.1 Discoverability Explained 362
Discovery and Interpretation, Discoverability and Interpretability in Abstract 364
Origins of Discovery 367
12.2 Profiling this Principle 368
12.3 Types of Discovery and Discoverability Meta Information 371
Design-Time and Runtime Discovery 371
Discoverability Meta Information 373
Functional Meta Data 374
Quality of Service Meta Data 374
12.4 Measuring Service Discoverability 375
Fundamental Levels 375
Custom Rating System 376
12.5 Discoverability and Service Design 376
Service Discoverability and Service Modeling 377
Service Discoverability and Granularity 378
Service Discoverability and Policy Assertions 378
Service Discoverability and Service Models 378
How Service Discoverability Affects Other Principles 378

12.6 Risks Associated with Service Discoverability 381
Post-Implementation Application of Discoverability 381
Application of this Principle by Non-Communicative Resources 381
12.7 Case Study Example 382
Service Profiles (Functional Meta Information) 382
Related Quality of Service Meta Information 386

Chapter 13: Service Composability (Composition Member Design and Complex Compositions) 387

13.1 Composition Explained 388
Composition in Abstract 388
Origins of Composition 390
13.2 Profiling this Principle 392
13.3 Composition Concepts and Terminology 396
Compositions and Composition Instances 397
Composition Members and Controllers 398
Service Compositions and Web Services 401
Service Activities 402
Composition Initiators 403
Point-to-Point Data Exchanges and Compositions 405
Types of Compositions 406
13.4 The Complex Service Composition 407
Stages in the Evolution of a Service Inventory 407
Defining the Complex Service Composition 410
Preparing for the Complex Service Composition 411
13.5 Measuring Service Composability and Composition Effectiveness Potential 412
Evolutionary Cycle States of a Composition 412
Composition Design Assessment 413
Composition Runtime Assessment 415
Composition Governance Assessment 417
Measuring Composability 419

13.6 Composition and Service Design 427
Service Composability and Granularity 427
Service Composability and Service Models 428
Service Composability and Composition Autonomy 430
Service Composability and Orchestration 430
How Service Composability Affects Other Principles 432
13.7 Risks Associated with Service Composition 437
Composition Members as Single Points of Failure 437
Composition Members as Performance Bottlenecks 437
Governance Rigidity of "Over-Reuse" in Compositions 438
13.8 Case Study Example 439

PART III: SUPPLEMENTAL

Chapter 14: Service-Orientation and Object-Orientation: A Comparison of Principles and Concepts 445

14.1 A Tale of Two Design Paradigms 446
14.2 A Comparison of Goals 449
Increased Business Requirements Fulfillment 450
Increased Robustness 451
Increased Extensibility 451
Increased Flexibility 452
Increased Reusability and Productivity 452
14.3 A Comparison of Fundamental Concepts 453
Classes and Objects 453
Methods and Attributes 454
Messages 454
Interfaces 456
14.4 A Comparison of Design Principles 457
Encapsulation 458
Inheritance 459

Generalization and Specialization 461
Abstraction 463
Polymorphism 463
Open-Closed Principle (OCP) 465
Don't Repeat Yourself (DRY) 465
Single Responsibility Principle (SRP) 466
Delegation 468
Association 469
Composition 470
Aggregation 471

14.5 Guidelines for Designing Service-Oriented Classes 472
Implement Class Interfaces 473
Limit Class Access to Interfaces 473
Do Not Define Public Attributes in Interfaces 473
Use Inheritance with Care 473
Avoid Cross-Service "has-a" Relationships 474
Use Abstract Classes for Modeling, Not Design 474
Use Façade Classes 474

Chapter 15: Supporting Practices 477

15.1 Service Profiles 478
Service-Level Profile Structure 478
Capability Profile Structure 480
Additional Considerations 482

15.2 Vocabularies 483
Service-Oriented Computing Terms 484
Service Classification Terms 484
Types and Associated Terms 485
Design Principle Application Levels 487

15.3 Organizational Roles 488
Service Analyst 490
Service Architect 490
Service Custodian 491
Schema Custodian 491
Policy Custodian 492

Service Registry Custodian 492
Technical Communications Specialist 493
Enterprise Architect 493
Enterprise Design Standards Custodian (and Auditor) 494

Chapter 16: Mapping Service-Orientation Principles to Strategic Goals 497

16.1 Principles that Increase Intrinsic Interoperability 498
16.2 Principles that Increase Federation 501
16.3 Principles that Increase Vendor Diversification Options . 501
16.4 Principles that Increase Business and Technology Domain Alignment 502
16.5 Principles that Increase ROI 504
16.6 Principles that Increase Organizational Agility 505
16.7 Principles that Reduce the Overall Burden of IT 507

PART IV: APPENDICES

Appendix A: Case Study Conclusion 513

Appendix B: Process Descriptions 517

B.1 Delivery Processes 518
Bottom-Up vs. Top-Down 518
The Inventory Analysis Cycle 520
Inventory Analysis and Service-Oriented Design 521
Choosing a Delivery Strategy 521
B.2 Service-Oriented Analysis Process 522
Define Analysis Scope 522
Identify Affected Systems 523
Perform Service Modeling 523

B.3 Service Modeling Process 523
B.4 Service-Oriented Design Processes. 525
Design Processes and Service Models 526
Service Design Processes and Service-Orientation 527

Appendix C: Principles and Patterns Cross-Reference 529

Additional Resources 533

About the Author 535

About the Photos 537

Index 539

Chapter 3

Service-Oriented Computing and SOA

3.1 Design Fundamentals

3.2 Introduction to Service-Oriented Computing

3.3 Goals and Benefits of Service-Oriented Computing

3.4 Case Study Background

One of the most challenging aspects of writing about or discussing technology is using industry terminology. Many IT terms suffer from wide-spread ambiguity, which sometimes makes having even the simplest conversation difficult. Take IT professionals from different organizations, put them in the same room, and you'll very likely hear questions like, "What exactly do you mean by component?" or "What is your definition of service?" or, my personal favorite, "What kind of SOA are you referring to?"

Fortunately, the primary subject matter of this book is very clear. We describe a distinct approach to designing solution logic. To ensure that the descriptions of associated topics are easily understood, a communications framework needs to be established, comprised of a collection of terms with very explicit definitions. That is what this chapter is dedicated to providing.

3.1 Design Fundamentals

Before we can begin exploring the details of service-oriented computing, we first need to establish some basic design terminology. The books in this series use a common vocabulary comprised of the following design-related terms:

- Design Characteristic
- Design Principle
- Design Paradigm
- Design Pattern
- Design Pattern Language
- Design Standard
- Best Practice

Depending on your sources, you will find differing definitions for these terms. More often than not, though, you will notice that they all are somewhat intertwined. The following sections explain each term and conclude with a section that illustrates how they form a common, fundamental design framework.

Design Characteristic

A characteristic of something is simply an attribute or quality. An automated business solution will have numerous unique characteristics that were established during its initial design (Figure 3.1). Hence, the type of *design characteristic* we are interested in is a specific attribute or quality of a body of solution logic that we document in a design specification and plan to realize in development.

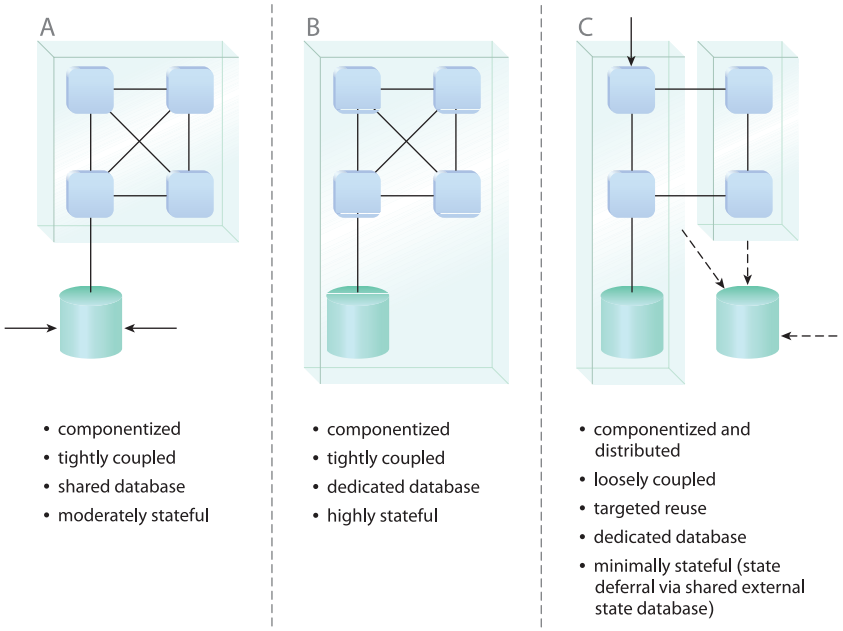


Figure 3.1
In this simple example, three distinct application designs (A, B, C) are established, each with its own list of design characteristics. We will continue to reference these applications in the upcoming sections. (Note that the small squares represent units of solution logic, solid arrows represent reuse or shared access, and dashed arrows represent state data transfer.)

Service-orientation emphasizes the creation of very specific design characteristics, while also *de*-emphasizing others. It is important to note that almost every design characteristic we explore is attainable to a certain *measure*. This means that it is generally not about whether solution logic does or does not have a certain characteristic; it is almost always about the extent to which a characteristic can or should be realized.

Although each system can have its own unique characteristics, we are primarily interested in establishing *common* design characteristics. Increased commonality ensures an

increased degree of consistency, making different kinds of solution logic more alike. When things are more alike they become more predictable. In the world of distributed, shareable logic, predictability is a good thing. Predictable design characteristics lead to predictable behavior. This, in turn, leads to increased reliability and the opportunity to leverage solution logic in many different ways.

Much of this book is dedicated to providing a means of establishing a specific collection of design characteristics that spread consistency, predictability, and reliability on many levels and for different purposes.

Design Principle

A principle is a generalized, accepted industry practice. In other words, it's something others are doing or promoting in association with a common objective. You can compare a principle with a best practice in that both propose a means of accomplishing something based on past experience or industry-wide acceptance.

When it comes to building solutions, a *design principle* represents a highly recommended guideline for shaping solution logic in a certain way and with certain goals in mind (Figure 3.2). These goals are usually associated with establishing one or more specific design characteristics (as a result of applying the principle).

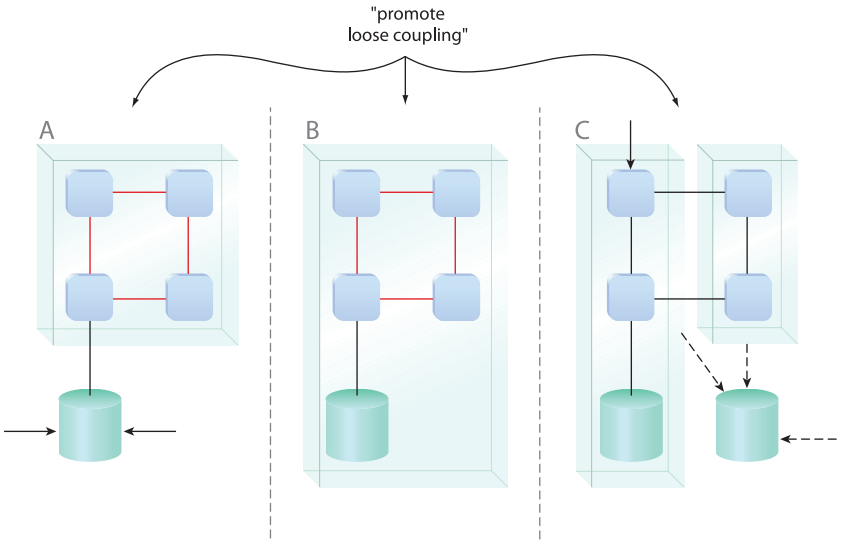


Figure 3.2
The repeated application of design principles increases the amount of common design characteristics. In this case, the coupling between solution logic units A and B has been loosened (as indicated by a reduction of connection points).

For example, we can have a principle as fundamental as one that states that solution logic should be distributable. Applying this principle results in the solution logic being partitioned into individually distributable units. This then establishes the distinct design characteristic of the solution logic becoming componentized. This is not only an example of a very broad design principle, but it is also the starting point for service-orientation.

The eight design principles documented in this book provide rules and guidelines that help determine exactly how solution logic should be decomposed and shaped into distributable units. A study of these principles further reveals what design characteristics these units should have to be classified as "quality" services capable of fulfilling the vision and goals associated with SOA and service-oriented computing.

Design Paradigm

There are many meanings associated with the term "paradigm." It can be an approach to something, a school of thought regarding something, or a combined set of rules that are applied within a predefined boundary.

A *design paradigm* within the context of business automation is generally considered a governing approach to designing solution logic. It normally consists of a set of complementary rules or principles that collectively define the overarching approach represented by the paradigm (Figure 3.3).

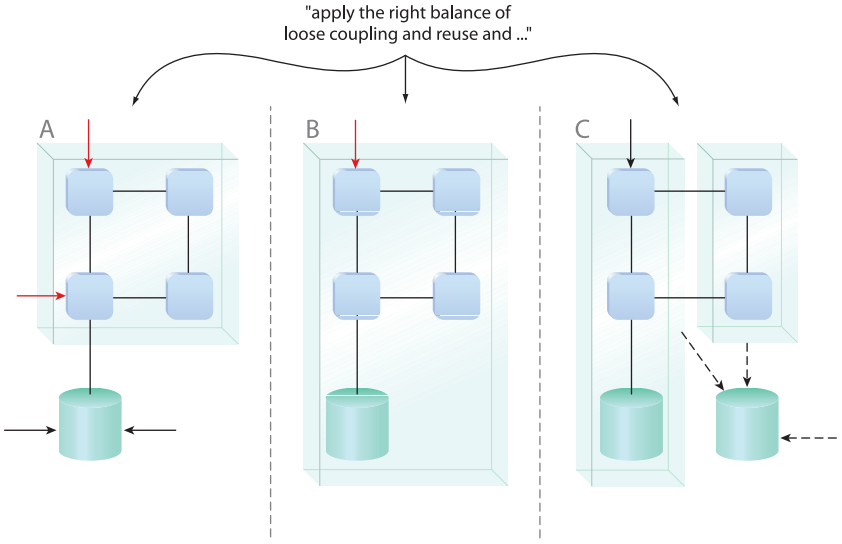


Figure 3.3
Because a design paradigm represents a collection of design principles, it further increases the degree of commonality across different bodies of solution logic. In the example, the amount of reuse in A and B has increased.

Object-orientation (or object-oriented design) is a classic example of an accepted design paradigm. It provides a set of principles that shape componentized solution logic in certain ways so as to fulfill a specific set of goals.

Along those very same lines, service-orientation represents its own distinct design paradigm. Like object-orientation, it is a paradigm that applies to distributed solution logic. However, because some of its principles differ from those associated with object-orientation (as explained in Chapter 14), it can result in the creation of different types of design characteristics.

Design Pattern

We've established that service-orientation is a design paradigm comprised of a set of design principles, each of which provides a generalized rule or guideline for realizing certain design characteristics. The paradigm itself sounds pretty complete, and it actually is. However, to successfully apply it in the real world requires more than just a theoretical understanding of its principles.

Service designers will be regularly faced with obstacles and challenges when attempting to apply a design paradigm because the realization of desired design characteristics is frequently complicated by various factors, including:

- Constraints imposed by the technology being used to build and/or host the units of solution logic.
- Constraints imposed by technology or systems that reside alongside the deployed units of solution logic.
- Constraints imposed by the requirements and priorities of the project delivering the units of solution logic.

A *design pattern* describes a common problem and provides a corresponding solution (Figure 3.4). It essentially documents the solution in a generic template format so that it can be repeatedly applied. Knowledge of design patterns not only arms you with an understanding of the potential problems designs may be subjected to, it provides answers as to how these problems are best dealt with.

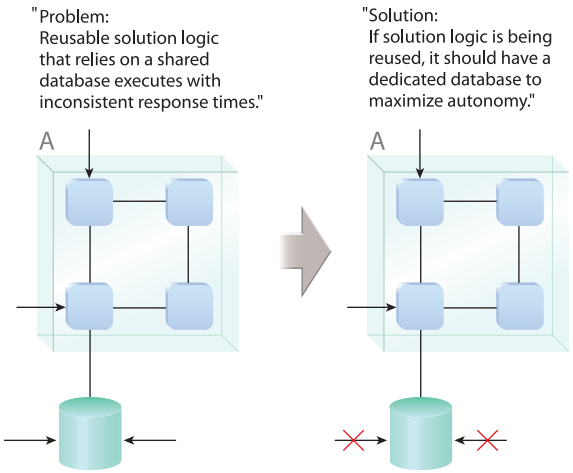


Figure 3.4
Patterns provide recommended solutions for common design problems. In this simplified example, a pattern suggests we reduce external access to a database to increase application autonomy.

Design patterns are born out of experience. Pioneers in any field had to undergo cycles of trial and error and by learning from what didn't work, approaches that finally did achieve their goals were developed. When a problem and its corresponding solution were identified as sufficiently common, the basis of a design pattern was formed. Design patterns can be further combined into compound patterns that solve larger problems and a series of patterns can form the basis of a pattern language, as explained next.

NOTE

Appendix C provides cross-references of design principles and associated design patterns documented as part of the pattern catalog published in *SOA: Design Patterns*.

Design Pattern Language

The application of one design pattern can raise new issues or problems for which another pattern may be required. A collection of related patterns can establish a formalized expression of a design process whereby each addresses a primary decision point. Combining patterns in this manner forms the basis of a *pattern language*.

A pattern language is essentially comprised of a chain of related design patterns that establish a configurable sequence in which the patterns can be applied (Figure 3.5). Such a language provides a highly effective means of communicating fundamental aspects of a given design approach because it supplies detailed documentation of each major step in a design process that shapes the design characteristics of solution logic.

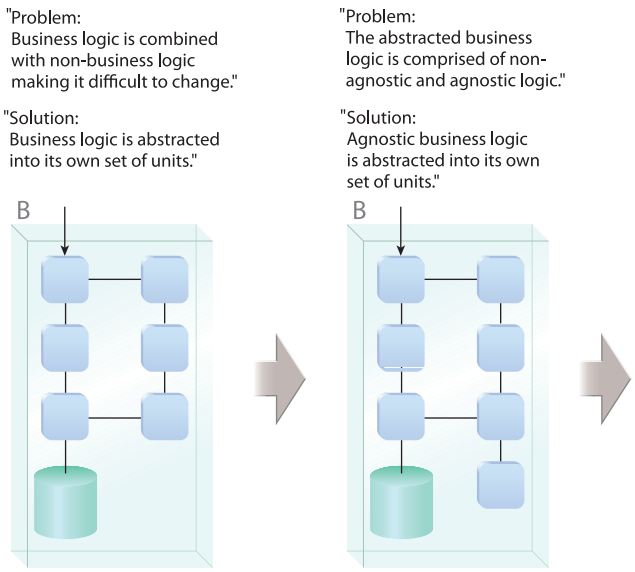


Figure 3.5
A sequence of related design patterns formalize the primary decision points of a design paradigm. In this example, the logic in application design B is decomposed as a result of one pattern, and then further decomposed as a result of another. Subsequent fundamental patterns continue to shape the logic.

NOTE

The fundamental paradigm and underlying philosophies of service-orientation and SOA are expressed through a basic pattern language as part of *SOA: Design Patterns*.

Design Standard

For an organization to successfully apply a design paradigm, it will require more than an adherence to the associated design principles and a knowledge of supporting design patterns. Every organization will have unique strategic goals and unique enterprise environments. These form a distinct set of requirements and constraints that need to be accommodated within solution designs.

Design standards are (usually mandatory) design conventions customized to consistently pre-determine solution design characteristics in support of organizational goals and optimized for specific enterprise environments. It is through the use of internal design standards that organizations can consistently deliver solutions tailored to their environments, resources, goals, and priorities (Figure 3.6).

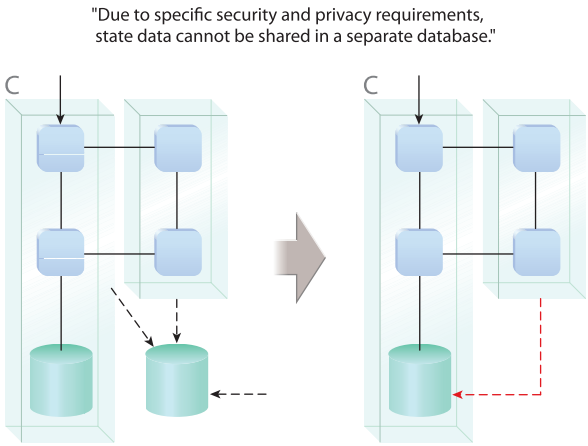


Figure 3.6
In this case, a design standard requires that C's original design be altered to remove access to a shared, external state database.

As with design principles, the application of design standards results in the creation of specific design characteristics. As with design patterns, design standards foster and refine these characteristics to avoid potential problems and to strengthen the overall solution design. In fact, it is recommended for design standards to be based upon or even derived from industry design principles and patterns.

Can you have design standards without design principles? Yes, it is actually common to have many design standards. Only some may need to relate back to principles in order to see through the application of the overall design paradigm. Different design standards may also be created to simply support other goals or compensate for constraints imposed by specific environmental, cultural, or technology-related factors. Although some standards may have no direct association with accepted design principles, there should always be an effort to keep all standards in relative alignment.

Can you have design principles without design standards? It usually depends on how committed an organization is to the governing design paradigm. If it sees potential in only using a subset of the paradigm's principles, then some principles may not be supported by corresponding design standards. However, this approach is not common.

Essentially, as with design principles, through standardization we want to build consistency into specific design characteristics—consistency in the quality of the characteristics and in how frequently they are implemented.

NOTE

One point of clarification often worth making when discussing standards is the difference between design standards and *industry standards*. The former, as we just described, refers to internal or custom standards that apply to the design of solution logic and systems for a particular enterprise. The latter generally represents open technology standards, such as those that comprise the XML and Web services platforms.

Sometimes organizations assume that if they use industry standards, they will end up with a standardized IT enterprise. While those XML and Web services specifications that have become ratified and accepted industry standards do establish a level of technology standardization, it is still up to an organization to consistently position and apply these technologies. Without design standards, industry standards can easily fail in achieving their potential.

Best Practice

A *best practice* is generally considered a technique or approach to solving or preventing certain problems (Figure 3.7). It is usually a practice that has industry recognition and has emerged from past industry experience.

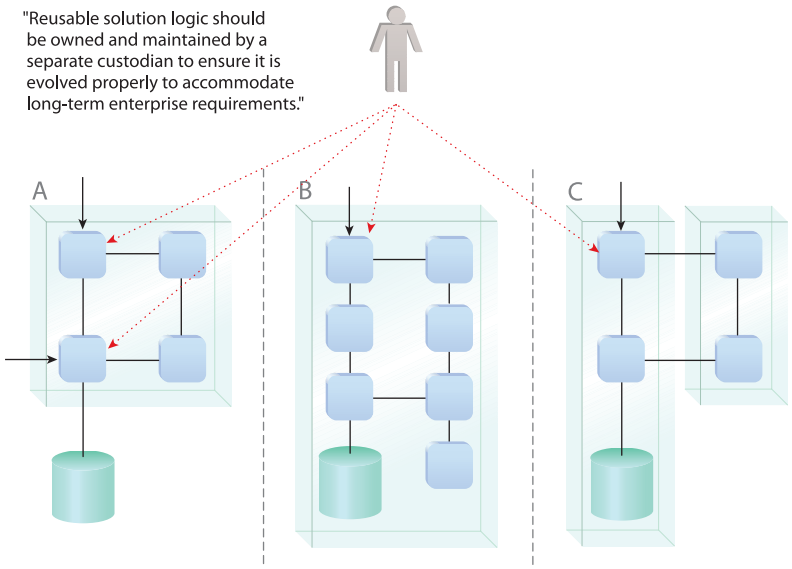


Figure 3.7
Best practices provide guidance in the form of general "lessons learned." In the example, it is suggested that the on-going maintenance of reusable solution logic units from all applications fall under a single custodian.

How then is a best practice differentiated from a design principle? In this book we make a clear distinction in that a design principle is limited to design only. A best practice can relate

to anything from project delivery to organizational issues, governance, or process. A design principle could be considered a best practice associated only with solution design.

Note that several best practices are provided throughout this book in support of applying design principles. An additional set of more detailed practices is located in Chapter 15.

A Fundamental Design Framework

Each of the previous sections described a piece of intelligence that can act as input into an overall design process. When designing service-oriented solutions it is practically inevitable that some or all of these pieces be used together. It is therefore important to understand how they relate to each other so that we can gain a foreknowledge of how and where they are best utilized.

Figure 3.8 shows how some of the more common parts of a design framework typically inter-relate and highlights how central the use of design principles can be. Figure 3.9 expands on this perspective by illustrating how the use of design patterns can further support and extend a basic design framework. Finally, Figure 3.10 shows how the parts of a design framework can ultimately help realize the application of the overarching design paradigm.

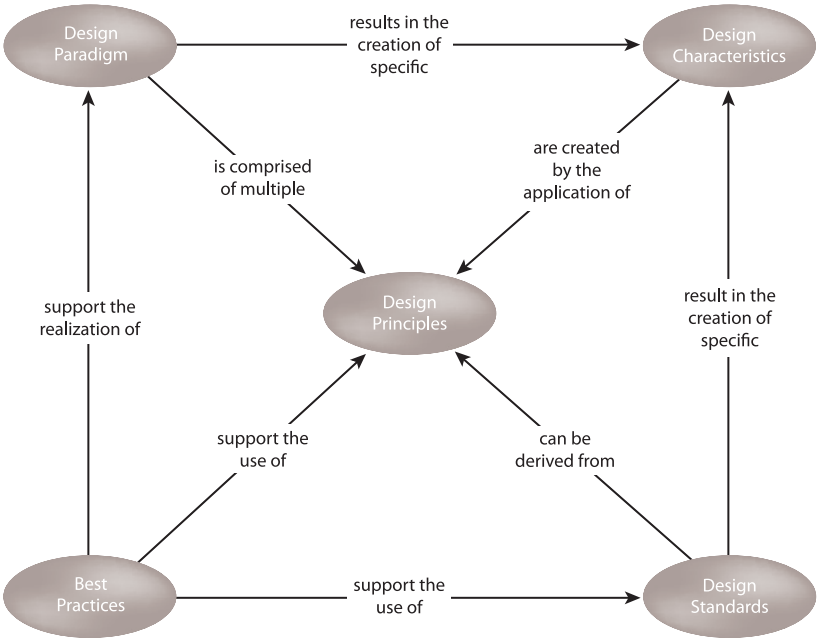


Figure 3.8

Fundamental design terms establish a basic taxonomy used throughout the upcoming chapters. This diagram hints at how some parts of a basic design framework can relate to each other.

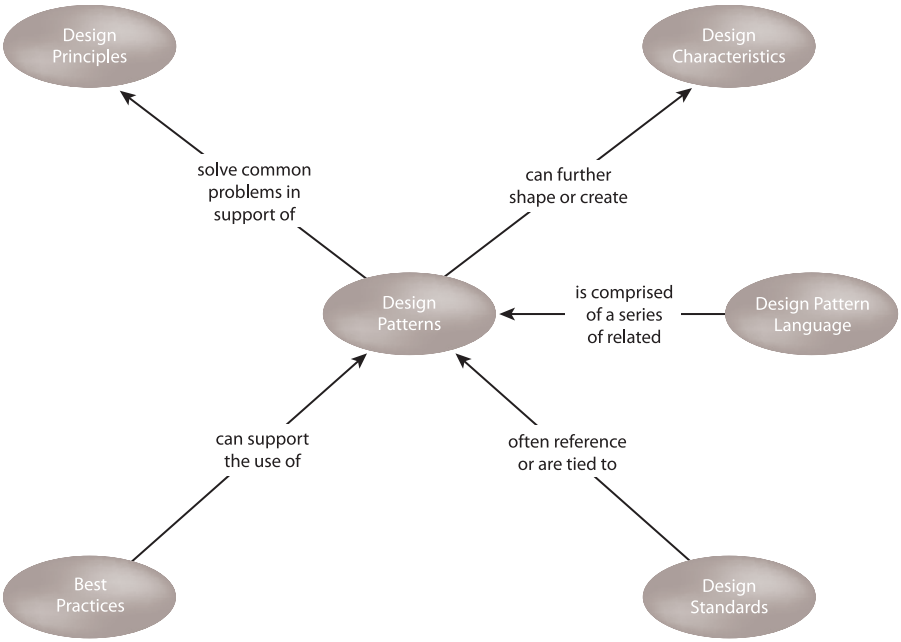


Figure 3.9

Design patterns provide additional intelligence that can enrich a design framework with a collection of proven solutions to common problems.

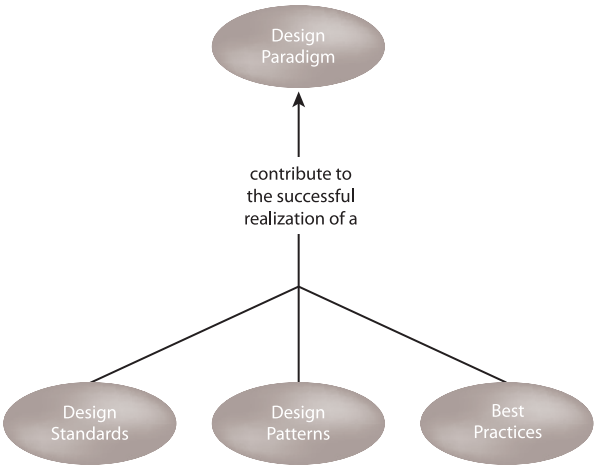


Figure 3.10

The purpose of applying a design paradigm is the achievement of certain goals. It is important to emphasize how design standards, design patterns, and best practices can all support the successful application of a design paradigm and, as a result, the attainment of its goals.

SUMMARY OF KEY POINTS

- A design principle is an accepted design guideline or practice that, when applied, results in the realization of specific design characteristics.
- A design paradigm represents a set of complementary design principles that are collectively applied in support of common goals.
- A design pattern identifies a common problem and provides a recommended solution.
- A design standard is a convention internal and specific to an enterprise that may or may not be derived from a design principle or pattern.

3.2 Introduction to Service-Oriented Computing

Service-oriented computing represents a new generation distributed computing platform. As such, it encompasses many things, including its own design paradigm and design principles, design pattern catalogs, pattern languages, a distinct architectural model, and related concepts, technologies, and frameworks.

It sounds like a pretty big umbrella, and it is. Service-oriented computing builds upon past distributed computing platforms and adds new design layers, governance considerations, and a vast set of preferred implementation technologies. That's why taking the time to understand its underlying mechanics *before* proceeding to the actual design and construction phases of a delivery project is time well spent.

To better appreciate the fundamental complexion of a typical service-oriented computing platform we need to describe each of its primary parts, which we'll refer to as *elements*:

- Service-Oriented Architecture
- Service-Orientation
- Service-Oriented Solution Logic
- Services
- Service Compositions
- Service Inventory

The following sections define each of these elements and conclude with a section that explains how they can inter-relate conceptually and physically. The basic symbols introduced in these sections are used repeatedly within subsequent parts of this book.

Service-Oriented Architecture

SOA establishes an architectural model that aims to enhance the efficiency, agility, and productivity of an enterprise by positioning services as the primary means through which solution logic is represented in support of the realization of strategic goals associated with service-oriented computing.

On a fundamental basis, the service-oriented computing platform revolves around the service-orientation design paradigm and its relationship with service-oriented architecture. In fact, the term "service-oriented architecture" and its associated acronym have been used so broadly by the media and within vendor marketing literature that it has almost become synonymous with service-oriented computing itself. It is therefore very important to make a clear distinction between what SOA actually is and how it relates to other service-oriented computing elements.

As a form of technology architecture, an SOA implementation can consist of a combination of technologies, products, APIs, supporting infrastructure extensions, and various other parts (Figure 3.11). The actual face of a deployed service-oriented architecture is unique within each enterprise; however it is typified by the introduction of new technologies and platforms that specifically support the creation, execution, and evolution of service-oriented solutions. As a result, building a technology architecture around the service-oriented architectural model establishes an environment suitable for solution logic that has been designed in compliance with service-orientation design principles.

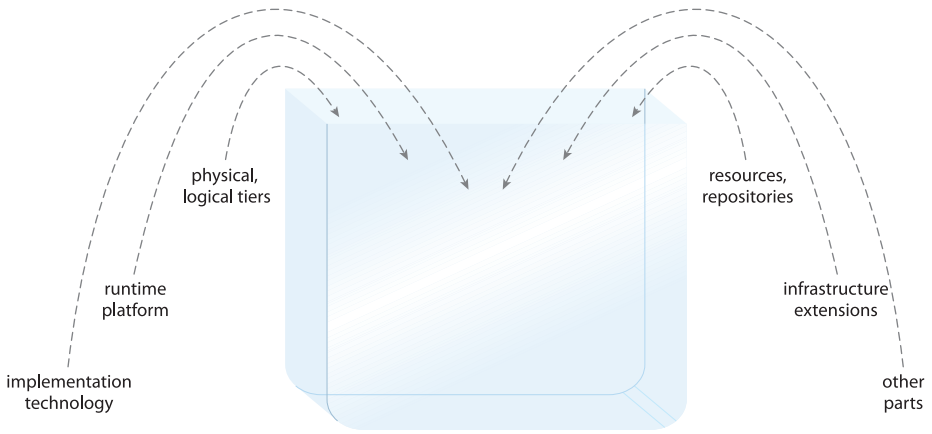


Figure 3.11
Container symbols are used to represent architectural implementation environments.

Service-Orientation, Services, and Service-Oriented Solution Logic

Service-orientation is a design paradigm comprised of a specific set of design principles. The application of these principles to the design of solution logic results in *service-oriented solution logic*. The most fundamental unit of service-oriented solution logic is the *service*.

Services exist as physically independent software programs with distinct design characteristics that support the attainment of the strategic goals associated with service-oriented computing. Each service is assigned its own distinct functional context and is comprised of a set of capabilities related to this context. Those capabilities suitable for invocation by external consumer programs are commonly expressed via a published service contract (much like a traditional API).

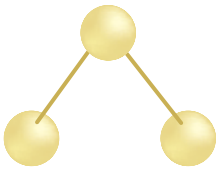
Figure 3.12 introduces the symbol used in this book to represent a service from an endpoint perspective. See the *SOA and Web Services* section for an introduction to the symbols used to illustrate a physical design perspective of services implemented as Web services. Note also that services and service-orientation are explored in detail in Chapter 4.

Figure 3.12
The yellow sphere symbol is used to represent a service. Alternatively, the chorded circle symbol introduced in Chapter 1 can also be used.

Service Compositions

A *service composition* is a coordinated aggregate of services. As explained in the *Effects of Service-Orientation on the Enterprise* section in Chapter 4, a composition of services (Figure 3.13) is comparable to a traditional application in that its functional scope is usually associated with the automation of a parent business process.

Figure 3.13
The symbol comprised of three connected spheres represents a service composition. Other, more detailed representations are based on the use of chorded circle symbols to illustrate which service capabilities are actually being composed.



The consistent application of service-orientation design principles leads to the creation of services with functional contexts that are agnostic to any one business process. These agnostic services are therefore capable of participating in multiple service compositions.

As further discussed in Chapters 13 and 16, the ability for a service to be naturally and repeatedly composable is fundamental to attaining several of the key strategic goals of service-oriented computing. Therefore, many of the design characteristics that distinguish a service enable it to effectively participate in service compositions.

Service Inventory

A *service inventory* is an independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise. Figure 3.14 establishes the symbol used to represent a service inventory in this book.

Figure 3.14
The service inventory symbol is comprised of yellow spheres within a blue container.

An IT enterprise may include a service inventory that represents the extent to which SOA has been adopted. Larger initiatives may even result in the enterprise in its entirety being comprised of an enterprise-wide service inventory. Alternatively, an enterprise environment can contain multiple service inventories, each of which can be individually standardized, governed, and supported by its own service-oriented technology architecture.

Service inventories are typically created through top-down delivery processes that result in the definition of service inventory blueprints. The subsequent application of service-orientation design principles and custom design standards throughout a service inventory is of paramount importance so as to establish a high degree of native inter-service interoperability. This supports the repeated, agile creation of effective service compositions. (Note that service inventory blueprints are explained later in this chapter.)

Understanding Service-Oriented Computing Elements

We'll be making reference to the previously defined elements throughout this book. Understanding them individually is just as important as understanding how they can relate to each other because these relationships establish some of the most fundamental dynamics of service-oriented computing.

Let's therefore revisit these elements with an emphasis on how each ties into others:

- *Service-oriented architecture* represents a distinct form of technology architecture designed in support of *service-oriented solution logic* which is comprised of *services* and *service compositions* shaped by and designed in accordance with *service-orientation*.

- *Service-orientation* is a design paradigm comprised of *service-orientation design principles*. When applied to units of solution logic, these principles create *services* with distinct design characteristics that support the overall goals and vision of *service-oriented computing*.
- *Service-oriented computing* represents a new generation computing platform that encompasses the *service-orientation paradigm* and *service-oriented architecture* with the ultimate goal of creating and assembling one or more *service inventories*.

These relationships are further illustrated in Figure 3.15.

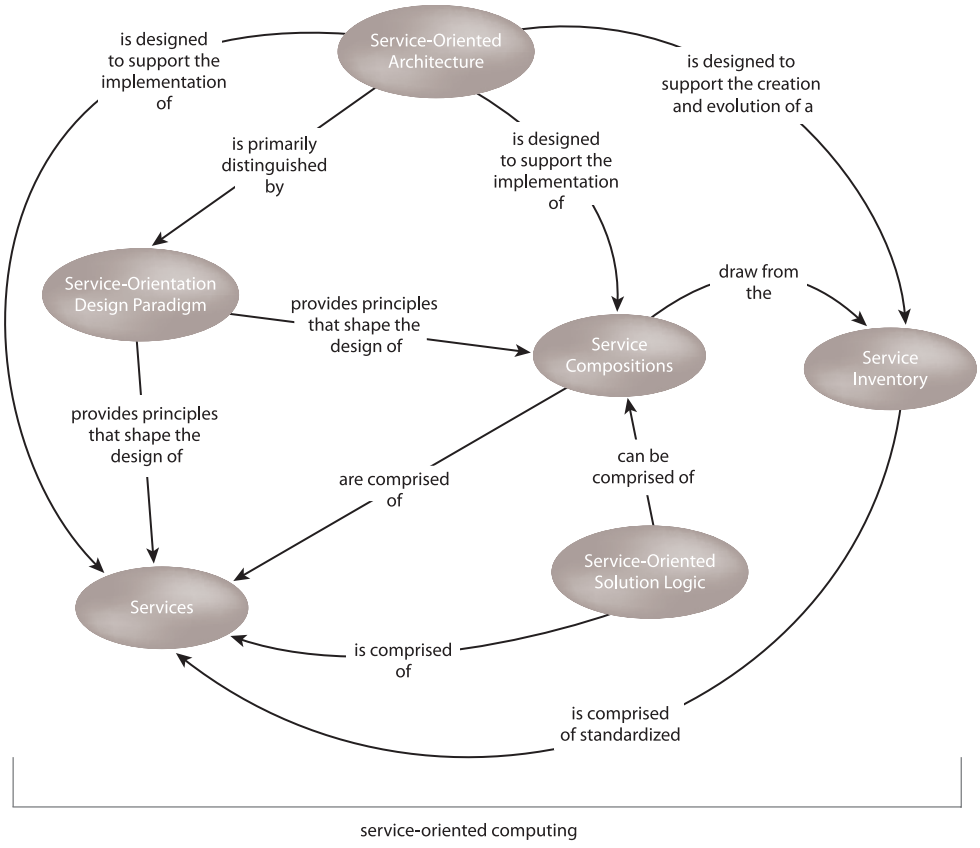


Figure 3.15
A conceptual view of how the elements of service-oriented computing can inter-relate.

To fully appreciate how these elements are ultimately used we need to explore how they translate into the real world. To do so, we need to clearly distinguish the role and position of each element within a physical implementation perspective, as follows:

- *Service-oriented solution logic* is implemented as *services* and *service compositions* designed in accordance with *service-orientation design principles*.
- A *service composition* is comprised of *services* that have been assembled to provide the functionality required to automate a specific business task or process.
- Because *service-orientation* shapes many *services* as agnostic enterprise resources, one *service* may be invoked by multiple consumer programs, each of which can involve that same *service* in a different *service composition*.
- A collection of standardized *services* can form the basis of a *service inventory* that can be independently administered within its own physical deployment environment.
- Multiple business processes can be automated by the creation of *service compositions* that draw from a pool of existing agnostic *services* that reside within a *service inventory*.
- *Service-oriented architecture* is a form of technology architecture optimized in support of *services*, *service compositions*, and *service inventories*.

This implementation-centric view brings to light how service-oriented computing can change the overall complexion of an enterprise. Because the majority of services delivered are positioned as reusable resources agnostic to business processes, they do not belong to any one application silo. By dissolving boundaries between applications, the enterprise is increasingly represented by a growing body of services that exist within an expanding service inventory (Figure 3.16).

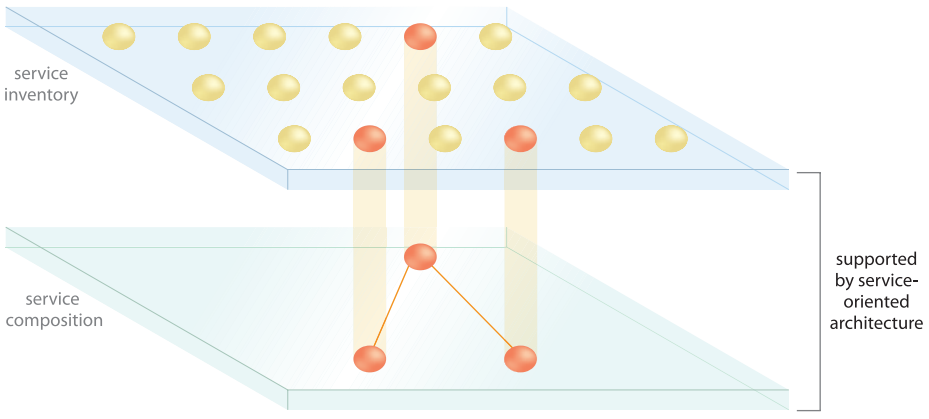


Figure 3.16
A service inventory establishes a pool of services, many of which will be deliberately designed to be reused within multiple service compositions.

NOTE

So far, an introductory perspective of service-oriented computing and its key elements has been established. However, when making reference to the service-oriented computing platform, we need to acknowledge the vast amounts of vendor development and runtime technologies that comprise its technology landscape. It is the makeup of these platforms and their combined technology innovations that have helped drive the evolution of service-oriented computing in the mainstream IT industry.

Service Models

When building various types of services, it becomes evident that they can be categorized depending on:

- the type of logic they encapsulate
- the extent of reuse potential this logic has
- how this logic relates to existing domains within the enterprise

As a result, there are three common classifications that represent the primary *service models* referenced throughout this book:

- Entity Services
- Task Services
- Utility Services

The use of these service models results in the creation of logical service abstraction layers, as shown in Figure 3.17.

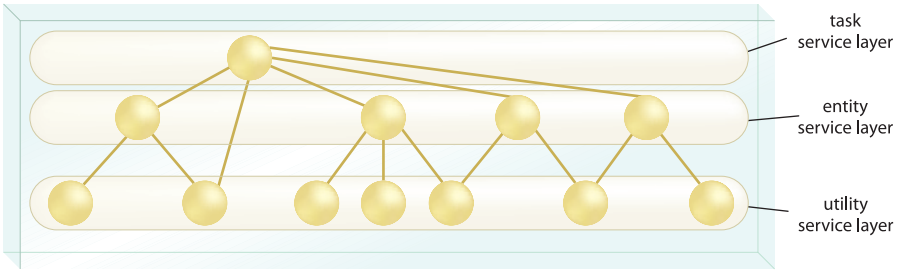


Figure 3.17

Common service abstraction layers established by service models, each of which is comprised of services shaped through the application of the service-orientation paradigm. Though these layers tend to form a natural composition hierarchy, there are no rules as to how services can be assembled.

Each of these three service models is further explained in the following sections.

Entity Services

In just about every enterprise, there will be business model documents that define the organization's relevant business entities. Examples of business entities include customer, employee, invoice, and claim. The *entity service* model (Figure 3.18) represents a business-centric service that bases its functional boundary and context on one or more related business entities. It is considered a highly reusable service because it is agnostic to most parent business processes. As a result, a single entity service can be leveraged to automate multiple parent business processes.

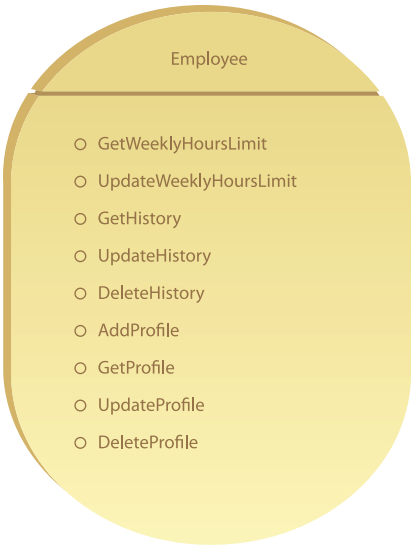


Figure 3.18
An example of an entity service. Several of its capabilities are reminiscent of traditional CRUD (create, read, update, delete) methods.

Entity services are also known as *entity-centric business services* or *business entity services*.

Task Services

A business service with a functional boundary directly associated with a specific parent business task or process is based on the *task service* model (Figure 3.19). This type of service tends to have less reuse potential and is generally positioned as the controller of a composition responsible for composing other, more process-agnostic services.

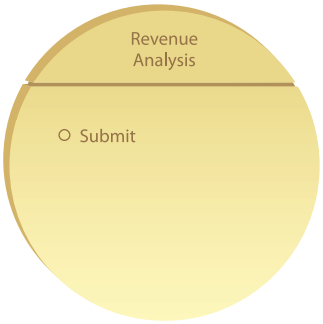


Figure 3.19
An example of a task service with a sole exposed capability required to initiate its encapsulated parent business process.

When discussing task services, one point of clarification often required is in relation to entity service capabilities. Each

capability essentially encapsulates business process logic in that it carries out a sequence of steps to complete a specific task. An entity Invoice service, for example, may have an Add capability that contains process logic associated with creating a new invoice record.

How then is what a task service encapsulates different from what an entity service's capabilities contain? The primary distinction has to do with the functional scope of the capability. The Invoice service's Add capability is focused solely on the processing of an invoice document. To carry out this process may require that the capability logic interact with other services representing different business entities, but the functional scope of the capability is clearly associated with the functional context of the Invoice service.

If, however, we had a billing consolidation process that retrieved numerous invoice and PO records, performed various calculations, and further validated consolidation results against client history billing records, we would have process logic that *spans* multiple entity domains and does not fit cleanly within a functional context associated with a business entity. This would typically constitute a "parent" process in that it consists of processing logic that needs to coordinate the involvement of multiple services.

Services with a functional context defined by a parent business process or task can be developed as standalone Web services or components—or—they may represent a business process definition hosted within an orchestration platform. In the latter case, the design characteristics of the service are somewhat distinct due to the specific nature of the underlying technology. In this case, it may be preferable to qualify the service model label accordingly. This type of service is referred to as the *orchestrated task service*.

Task services are also known as *task-centric business services* or *business process services*. Orchestrated task services are also known as *process services*, *business process services*, or *orchestration services*.

NOTE

There is a potential point of confusion when referring to these types of services as "business process services" or when renaming the task service layer to "business process layer." Just about every capability within every business service encapsulates an extent of business process logic. Establishing a task service layer does not abstract or centralize all business process logic. Its purpose is primarily to abstract non-agnostic process logic in support of agnostic service models. If there's a preference to incorporate the term "business process" within the title of this service layer, then it's recommended that it be further qualified with "parent" (as in the "parent business process layer").

Utility Services

Each of the previously described service models has a very clear focus on representing business logic. However, within the realm of automation, there is not always a need to associate logic with a business model or process. In fact, it can be highly beneficial to deliberately establish a functional context that is *non*-business-centric. This essentially results in a distinct, technology-oriented service layer.

The *utility service* model (Figure 3.20) accomplishes this. It is dedicated to providing reusable, cross-cutting utility functionality, such as event logging, notification, and exception handling. It is ideally application agnostic in that it can consist of a series of capabilities that draw from multiple enterprise systems and resources, while making this functionality available within a very specific processing context.

Figure 3.20
An example of a utility service providing a set of capabilities associated with proprietary data format transformation.

Utility services are also known as *application services*, *infrastructure services*, or *technology services*.

NOTE

Entity, task, and utility service models are intentionally generic in nature in that they apply to just about any type of enterprise. Customized variations can be further derived to fulfill specific types of domain abstraction.

SOA and Web Services

It is very important to view and position SOA as an architectural model that is agnostic to any one technology platform (Figure 3.21). By doing so, an enterprise is given the freedom to continually pursue the strategic goals associated with service-oriented computing by leveraging future technology advancements. In the current marketplace, the technology platform most associated with the realization of SOA is Web services.

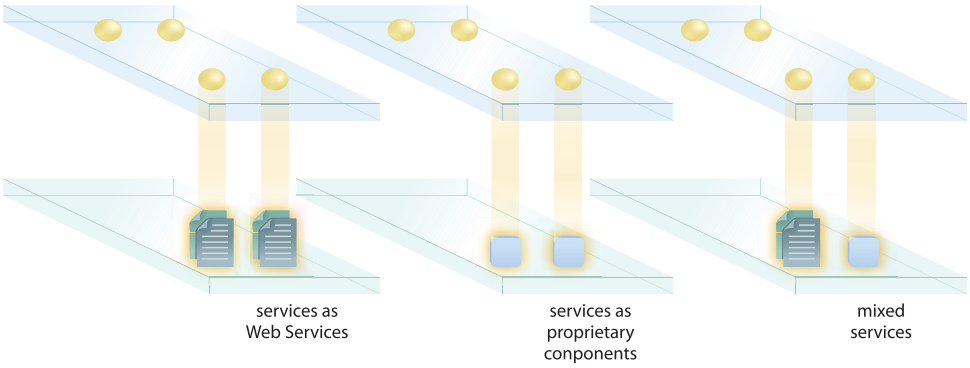


Figure 3.21
Service-oriented solutions can be comprised of services built as Web services, components, or combinations of both.

Web Services Standards

The Web services platform is defined through a number of industry standards that are supported throughout the vendor community. This platform can be partitioned into two clearly identifiable generations, each associated with a collection of standards and specifications:

- *First-Generation Web Services Platform*

  The original Web services technology platform is comprised of the following core open technologies and specifications: Web Services Description Language (WSDL), XML Schema Definition Language (XSD), SOAP (formerly the Simple Object Access Protocol), UDDI (Universal Description, Discovery, and Integration), and the WS-I Basic Profile.

  These specifications have been around for some time and have been adopted across the IT industry. However, the platform they collectively represent seriously lacks several of the quality of service features required to deliver mission critical, enterprise-level production functionality.

- *Second-Generation Web Services Platform (WS-* extensions)*

  Some of the greatest quality of service-related gaps in the first-generation platform lie in the areas of message-level security, cross-service transactions, and reliable messaging. These, along with many other extensions, represent the second-generation Web services platform. Consisting of numerous specifications that

build upon the fundamental first-generation messaging framework, this set of Web services technologies (generally labeled as "WS-*") provides a rich feature-set far more sophisticated both in technology and in design. An example of a WS-* standard referenced throughout this book is WS-Policy.

Web Services Architecture

A typical Web service is comprised of the following:

- A physically decoupled technical *service contract* consisting of a WSDL definition, an XML schema definition, and possibly a WS-Policy definition. This service contract exposes public functions (called operations) and is therefore comparable to a traditional application programming interface (API).
- A body of programming logic. This logic may be custom-developed for the Web service, or it may exist as legacy logic that is being wrapped by a Web service in order for its functionality to be made available via Web services communication standards. In the case that logic is custom-developed, it generally is created as components and is referred to as the *core service logic* (or *business logic*).
- *Message processing logic* that exists as a combination of parsers, processors, and service agents. Much of this logic is provided by the runtime environment, but it can also be customized. The programs that carry out message-related processing are primarily event-driven and therefore can intercept a message subsequent to transmission or prior to receipt. It is common for multiple message processing programs to be invoked with every message exchange.

A Web service can be associated with temporary *roles*, depending on its utilization at runtime. For example, it acts as a service provider when it receives and responds to request messages, but can also assume the role of service consumer when it is required to issue request messages to other Web services.

When Web services are positioned within service compositions, it is common for them to transition through service provider and service consumer roles (additional composition-related roles are explained in Chapter 13). Note also that regular programs, components, and legacy systems can also act as Web service consumers as long as they are able to communicate using Web services standards.

Figure 3.22 introduces the symbols used to illustrate physical representations of Web services in this book. Service-orientation principles can affect the design of all displayed parts.

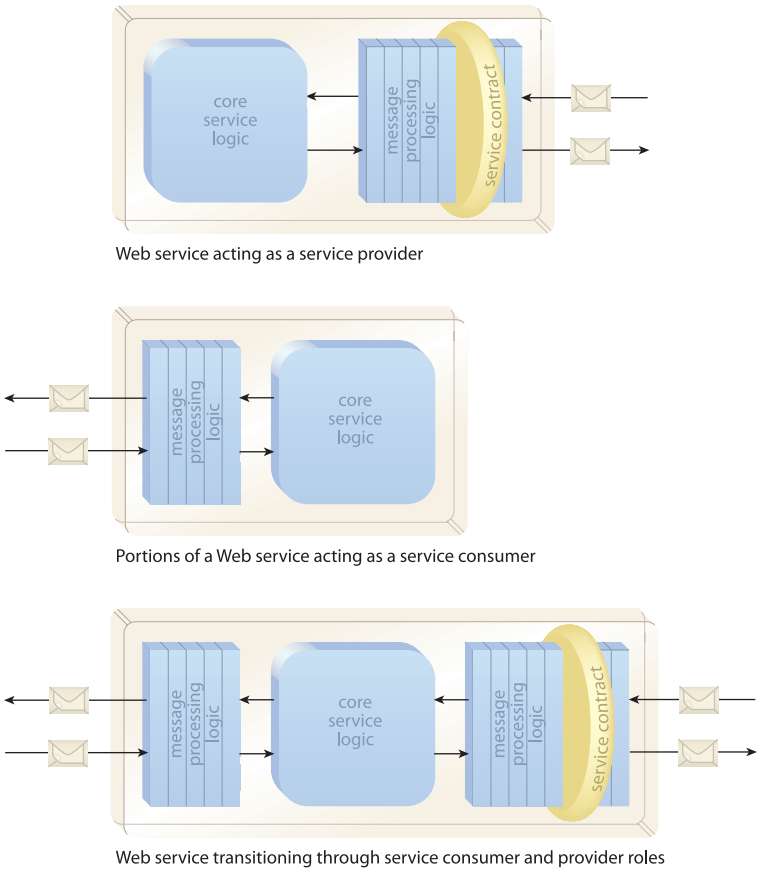


Figure 3.22
Three variations of a single Web service showing the different physical parts of its architecture that come into play, depending on the role it assumes at runtime.

Web Services and Service-Oriented Computing

The popularity of Web services preceded that of service-oriented computing. As a result, their initial use was primarily within traditional distributed solutions wherein they were most commonly used to facilitate point-to-point integration channels. As the maturity and adoption of Web services standards increased, so did the scope of their utilization.

With service-oriented computing comes a distinct architectural model that has been positioned by the vendor community as one that can fully leverage the open interoperability potential of Web services, especially when individual services are consistently shaped by service-orientation. For example, when exposing reusable logic as Web services, the reuse potential is significantly increased. Because service logic can now be accessed via a vendor-neutral communications framework, it becomes available to a wider range of service consumer programs.

Additionally, the fact that Web services provide a communications framework based on physically decoupled contracts allows each service contract to be fully standardized independently from its implementation. This facilitates a potentially high level of service abstraction while providing the opportunity to fully decouple the service from any proprietary implementation details. As explored in Part II, all of these characteristics are desirable when pursuing key principles, such as Standardized Service Contracts, Service Reusability, Service Loose Coupling, Service Abstraction, and Service Composability.

For example, transformation avoidance is a key goal of Standardized Service Contracts. As explained in Chapter 6, this principle advocates the standardization of the data model expressed by the service contract so as to increase intrinsic interoperability by reducing the need for transformation technologies. As illustrated in Figure 3.23, services delivered via disparate component platforms still require the transformation of technology regardless of whether data types are standardized. Services expressed through Web service contracts have the potential to avoid transformation altogether.

NOTE

To learn more about first and second-generation Web services technologies, read the tutorials posted at `www.ws-standards.com` or visit `www.soaspecs.com` and browse through the actual specifications. It is also important to acknowledge service communication mediums that provide an alternative to SOAP-based messaging, such as Representational State Transfer (REST) and Plain Old XML (POX). While these are not covered in this book, it would be worthwhile reading up on them to understand how they differ and where they are most commonly encountered.

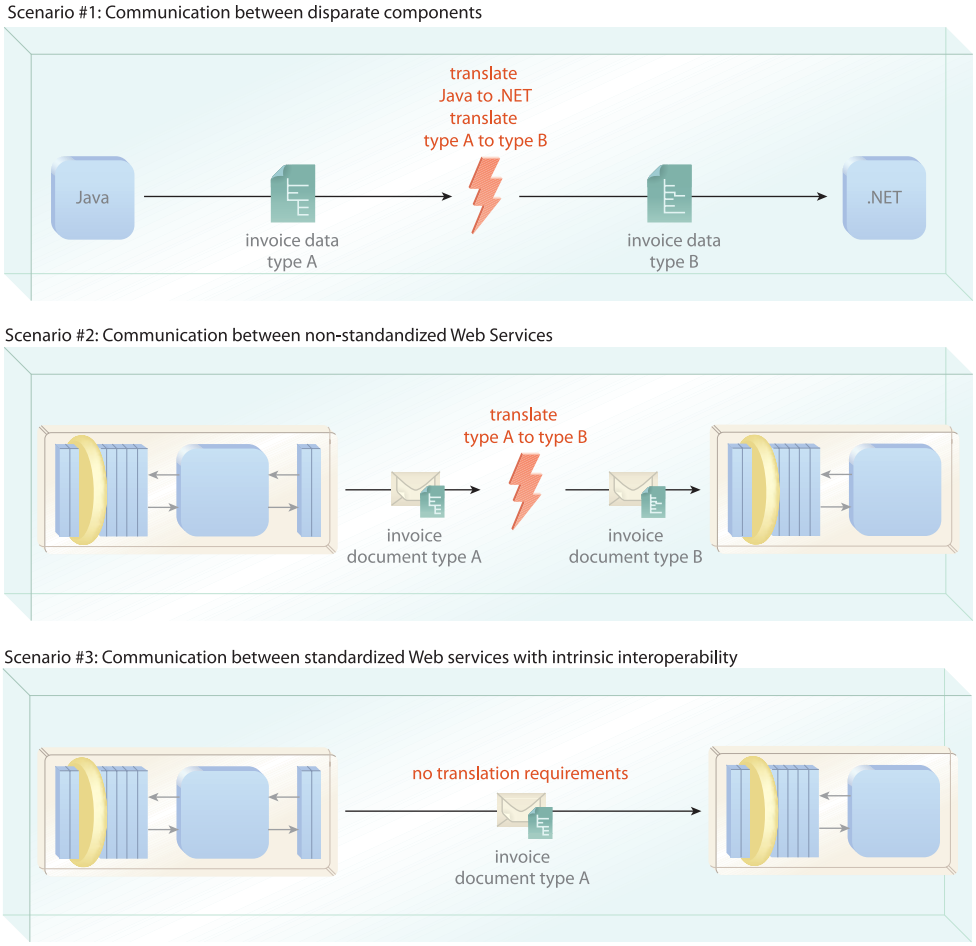


Figure 3.23
Three common data exchange scenarios demonstrating the effect of transformation avoidance.

Service Inventory Blueprints

An ultimate goal of an SOA transition effort is to produce a collection of standardized services that comprise a service inventory. The inventory can be structured into layers according to the service models used, but it is the application of the service-orientation paradigm to all services that positions them as valuable IT assets in full alignment with the strategic goals associated with the SOA project.

However, before any services are actually built, it is desirable to establish a conceptual blueprint of all the planned services for a given inventory. This perspective is documented in the *service inventory blueprint*. There are several common business and data models that, if they exist within an organization, can provide valuable input for this specification. Examples include business entity models, logical data models, canonical data and message models, ontologies, and other information architecture models.

A service inventory blueprint is also known as a *service enterprise model* or a *service inventory model*.

Service-Oriented Analysis and Service Modeling

To effectively deliver standardized services in support of building a service inventory, it is recommended that organizations adopt a methodology specific to SOA and consisting of structured analysis and design processes.

Within SOA projects, these processes are centered around the accurate expression of business logic through technology, which requires that business analysts play a more active role in defining the conceptual design of solution logic. This guarantees a higher degree of alignment between the documented business models and their implementation as services. Agnostic business services especially benefit from hands-on involvement of business subject matter experts, as the improved accuracy of their business representation increases their overall longevity once deployed.

Service-oriented analysis establishes a formal analysis process completed jointly by business analysts and technology architects. *Service modeling,* a sub-process of service-oriented analysis, produces conceptual service definitions called *service candidates*. Iterations through the service-oriented analysis and service modeling processes result in the gradual creation of a collection of service candidates documented as part of a service inventory blueprint.

While the collaborative relationship between business analysts and architects depicted at the lower half of Figure 3.24 may not be unique to an SOA project, the nature and scope of the analysis process is.

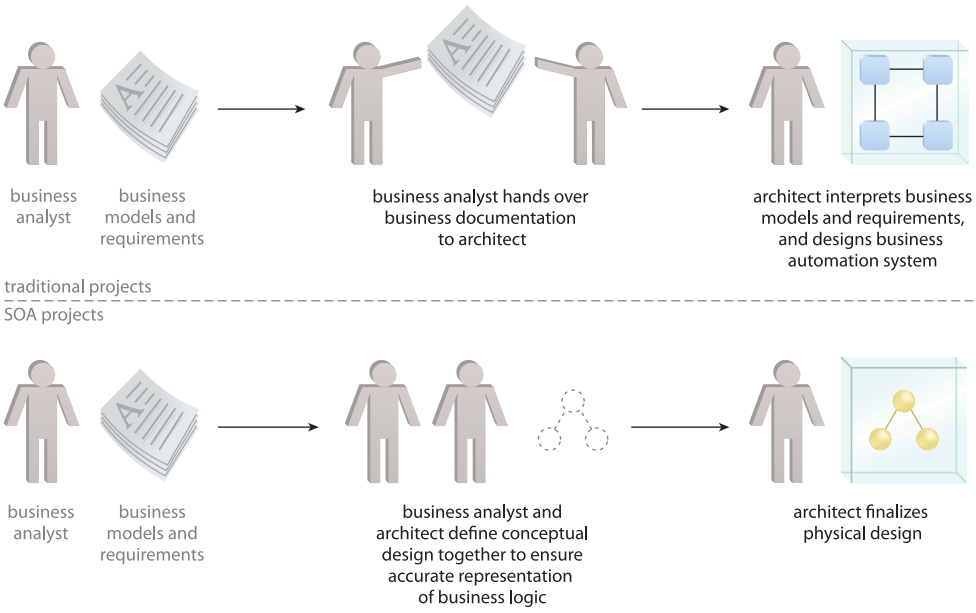


Figure 3.24
A look at how the collaboration between business analysts and technology architects changes with SOA projects.

Service-Oriented Design

The *service-oriented design* process uses a set of predefined service candidates from the service inventory blueprint as a starting point from which they are shaped into actual physical service contracts.

When carrying out service-oriented design, a clear distinction is made between service candidates and services. The former represents a conceptual service that has not been implemented, whereas the latter refers to a physical service.

As shown in Figure 3.25, the traditional (non-standardized) means by which Web service contracts are generated results in services that continue to express the proprietary nature of what they encapsulate. Creating the Web service contract *prior* to development allows for standards to be applied so that the federated endpoints established by Web services are consistent and aligned. This "contract first" approach lies at the heart of service-oriented design and has inspired separate design processes for services based on different service models.

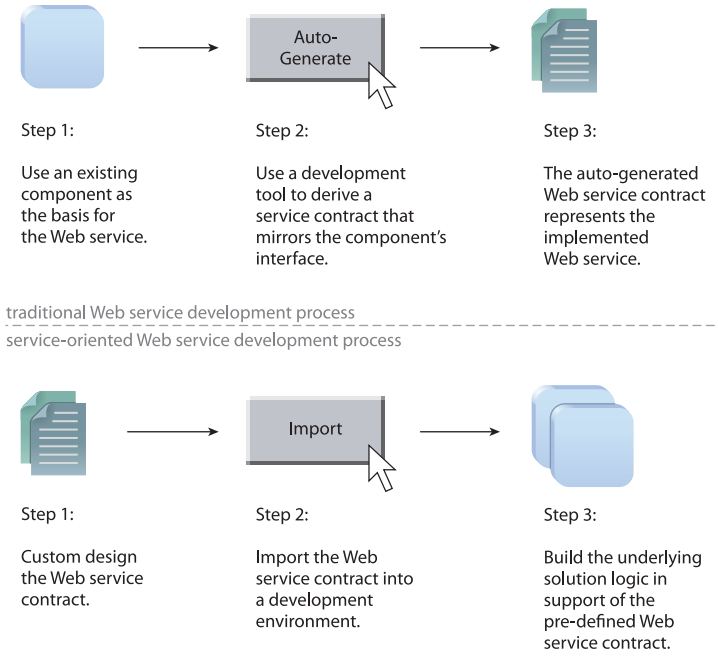


Figure 3.25
Unlike the popular process of deriving Web service contracts from existing components, SOA advocates a specific approach that encourages us to postpone development until after a custom designed, standardized contract is in place.

NOTE

Appendix B contains illustrations and brief descriptions of service-oriented analysis and design processes for reference purposes.

"Service-Oriented Architecture: Concepts, Technology, and Design"

Descriptions of first and second-generation Web services technologies, service models, service layers and variations of SOA, as well as a mainstream SOA methodology providing step-by-step process descriptions for service-oriented analysis, service modeling, and service-oriented design are explained in detail in the book *Service-Oriented Architecture: Concepts, Technology, and Design*. SOA is fundamental to all of the content in the remaining chapters and therefore a solid understanding of the concepts behind its architectural model and technologies commonly used for its implementation is recommended.

SUMMARY OF KEY POINTS

- The service-oriented computing platform is comprised of a distinct set of elements, each of which represents a specific aspect of service-oriented computing, and all of which are collectively applied to achieve its goals.
- Service models are used to establish service layers by categorizing services based on the type of logic they encapsulate.
- SOA represents an implementation-agnostic architectural model. However, Web services currently provide the foremost means of implementing services.

3.3 Goals and Benefits of Service-Oriented Computing

It is very important to establish why both vendor and end-user communities within the IT industry are going through the trouble of adopting the service-oriented computing platform and embracing all of the change that comes with it.

The vision behind service-oriented computing is extremely ambitious and therefore also very attractive to any organization interested in truly improving the effectiveness of its IT enterprise. A set of common goals and benefits has emerged to form this vision. These establish a target state for an enterprise that successfully adopts service-orientation.

The upcoming set of sections describe each of these strategic goals and benefits (also displayed in Figure 3.26):

- Increased Intrinsic Interoperability
- Increased Federation
- Increased Vendor Diversification Options
- Increased Business and Technology Domain Alignment
- Increased ROI
- Increased Organizational Agility
- Reduced IT Burden

It is beneficial to understand the significance of these goals and benefits prior to studying and applying service-orientation so that design principles are consistently viewed within a strategic context.

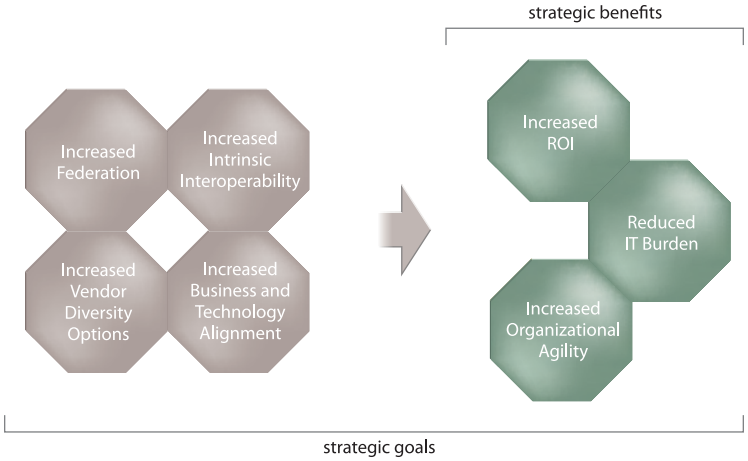


Figure 3.26

The seven identified goals are inter-related and can be further categorized into two groups: strategic goals and resulting benefits. Increased organization agility, increased ROI, and reduced IT burden are concrete benefits resulting from the attainment of the remaining four goals.

An important message of this book in general is that there is a concrete link between successfully applying service-orientation design principles and successfully attaining these specific goals and benefits (a point which is further detailed in Chapter 16).

NOTE

As previously explained, the term "SOA" has been used so much in the media and within marketing literature that it has become synonymous with what the entire service-oriented computing platform represents. Therefore, the goals and benefits listed here are frequently associated with SOA as well.

Increased Intrinsic Interoperability

Interoperability refers to the sharing of data. The more interoperable software programs are, the easier it is for them to exchange information. Software programs that are not interoperable need to be integrated. Therefore, integration can be seen as a process that enables interoperability. A goal of service-orientation is to establish native interoperability within services in order to reduce the need for integration (Figure 3.27). In fact, integration as a concept begins to fade within service-oriented environments (as further explained in the *Effects of Service-Orientation on the Enterprise* section in Chapter 4).

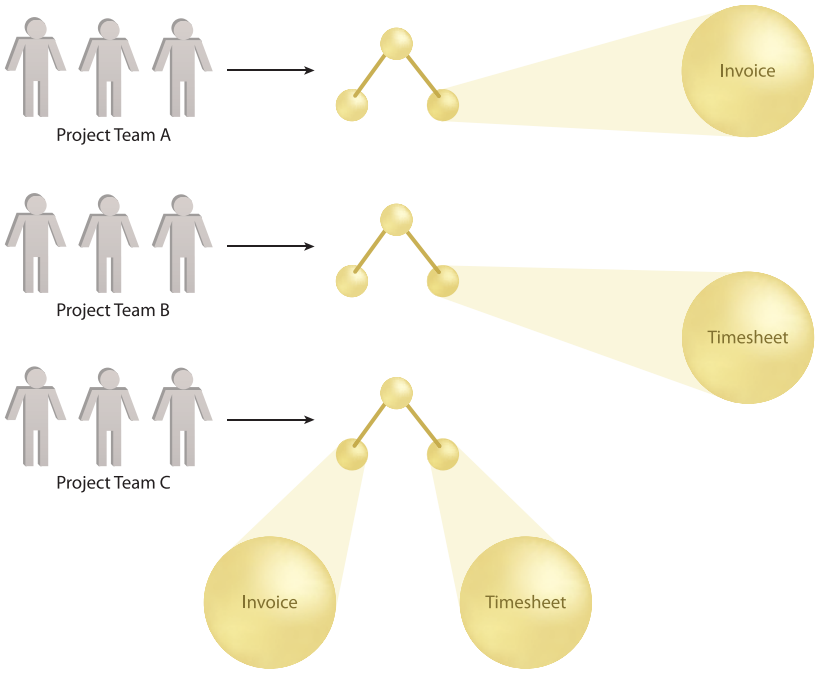


Figure 3.27
Services are designed to be intrinsically interoperable regardless of when and for which purpose they are delivered. In this example, the intrinsic interoperability of the Invoice and Timesheet services delivered by Project Teams A and B allow them to be combined into a new service composition by Project Team C.

Interoperability is specifically fostered through the consistent application of design principles and design standards. This establishes an environment wherein services produced by different projects at different times can be repeatedly assembled together into a variety of composition configurations to help automate a range of business tasks.

Intrinsic interoperability represents a fundamental goal of service-orientation that establishes a foundation for the realization of other strategic goals and benefits. Contract standardization, scalability, behavioral predictability, and reliability are just some of the design characteristics required to facilitate interoperability, all of which are addressed by the service-orientation principles documented in this book.

How specifically service-orientation design principles foster interoperability within services is explained in the *Service-Orientation and Interoperability* section of Chapter 4.

Increased Federation

A federated IT environment is one where resources and applications are united while maintaining their individual autonomy and self-governance. SOA aims to increase a federated perspective of an enterprise to whatever extent it is applied. It accomplishes this through the widespread deployment of standardized and composable services each of which encapsulates a segment of the enterprise and expresses it in a consistent manner.

In support of increasing federation, standardization becomes part of the extra up-front attention each service receives at design time. Ultimately this leads to an environment where enterprise-wide solution logic becomes naturally harmonized, regardless of the nature of its underlying implementation (Figure 3.28).

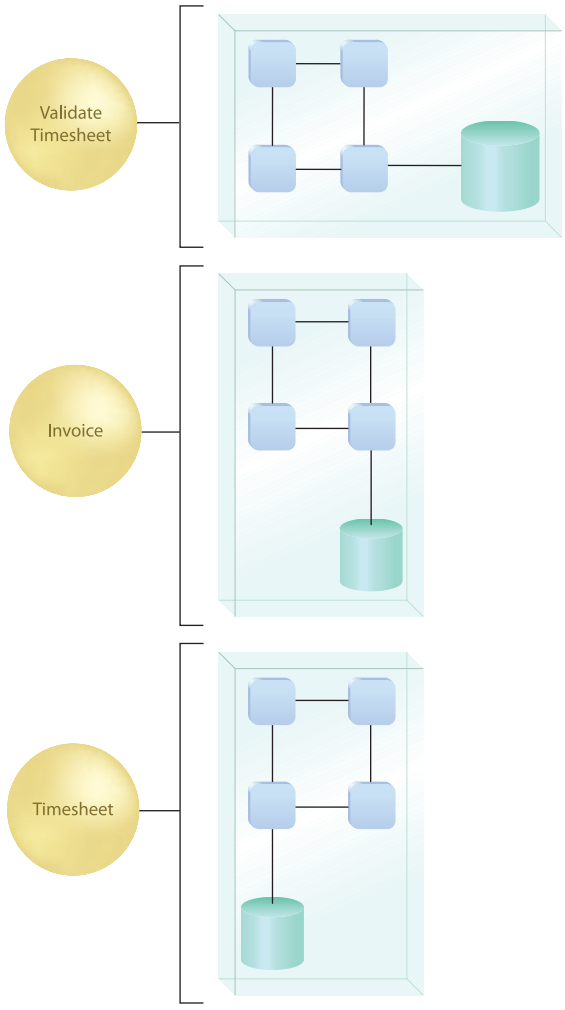


Figure 3.28
Three service contracts establishing a federated set of endpoints, each of which encapsulates a different implementation.

When service-oriented solutions are built via the Web services technology platform, the level of attainable federation is further elevated because services can leverage the non-proprietary nature of the technologies themselves. However, even when using Web services the key success factor to achieving true unity and federation remains the application of design principles and standards.

Increased Vendor Diversification Options

Vendor diversification refers to the ability an organization has to pick and choose "best-of-breed" vendor products and technology innovations and use them together within one enterprise. It is not necessarily beneficial for an organization to have a vendor-diverse environment; however, it is beneficial to have the *option* to diversify when required. To have and retain this option requires that its technology architecture not be tied or locked into any one specific vendor platform.

This represents an important state for an enterprise in that it provides the constant freedom for an organization to change, extend, and even replace solution implementations and technology resources without disrupting the overall, federated service architecture. This measure of governance autonomy is attractive because it prolongs the lifespan and increases the financial return of automation solutions.

By designing a service-oriented architecture in alignment with but neutral to major vendor SOA platforms and by positioning service contracts as standardized endpoints throughout a federated enterprise, proprietary service implementation details can be abstracted to establish a consistent inter-service communications framework. This provides organizations with constant options by allowing them to diversify their enterprises as needed (Figure 3.29).

Vendor diversification is further supported by taking advantage of the standards-based, vendor-neutral Web services framework. Because they impose no proprietary communication requirements, Web services further decrease dependency on vendor platforms. As with any other implementation medium, though, Web services need to be shaped and standardized through service-orientation in order to become a federated part of an SOA.

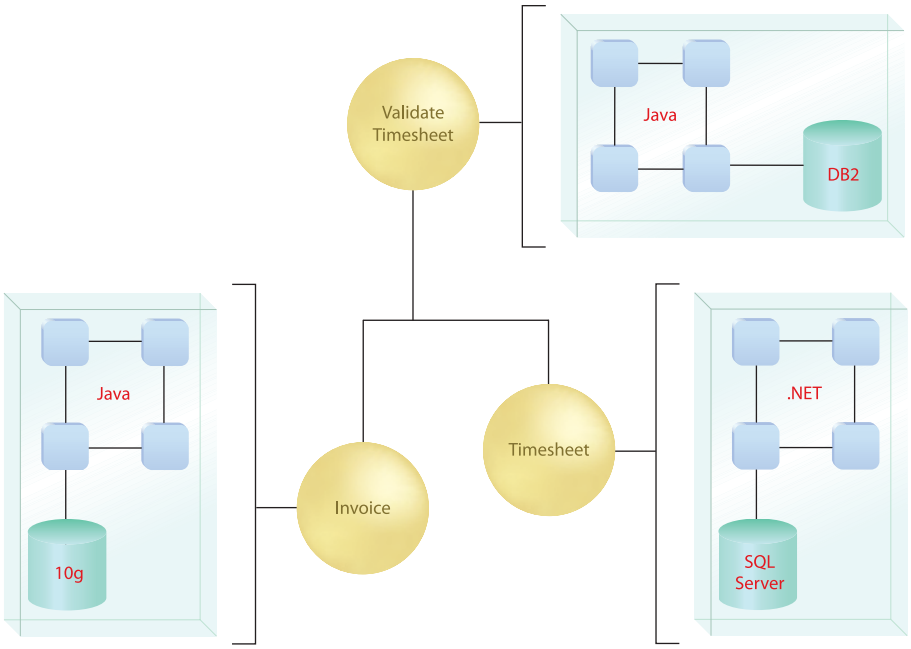


Figure 3.29
A service composition consisting of three services, each of which encapsulates a different vendor automation environment. If service-orientation is adequately applied to the services, underlying disparity will not inhibit their ability to be combined into effective compositions.

Increased Business and Technology Domain Alignment

The extent to which IT business requirements are fulfilled is often associated with the accuracy with which business logic is expressed and automated by solution logic. Although initial applications have traditionally been designed to address immediate and tactical requirements, it has historically been challenging to keep applications in alignment with business needs when the nature and direction of the business changes.

Service-oriented computing introduces a design paradigm that promotes abstraction on many levels. One of the most effective means by which functional abstraction is applied is the establishment of service layers that accurately encapsulate and represent business models. By doing so, common, pre-existing representations of business logic (business entities, business processes) can exist in implemented form as physical services.

This is accomplished by incorporating a structured analysis and modeling process that requires the hands-on involvement of business subject matter experts in the actual definition of the conceptual service candidates (as explained in the *Service-Oriented Analysis*

and Service Modeling section). The resulting service designs are capable of aligning automation technology with business intelligence on an unprecedented level (Figure 3.30).

Furthermore, the fact that services are designed to be intrinsically interoperable directly facilitates business change. As business processes are augmented in response to various factors (business climate changes, new competitors, new policies, new priorities, etc.) services can be reconfigured into new compositions that reflect the changed business logic. This allows a service-oriented technology architecture to evolve in tandem with the business itself.

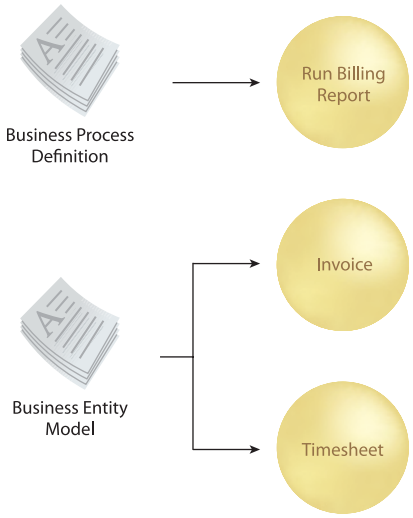


Figure 3.30
Services with business-centric functional contexts are carefully modeled to express and encapsulate corresponding business models and logic.

Increased ROI

Measuring the return on investment (ROI) of automated solutions is a critical factor in determining just how cost effective a given application or system actually is. The greater the return, the more an organization benefits from the solution. However, the lower the return, the more the cost of automated solutions eats away at an organization's budgets and profits.

Traditional, silo-based applications tend to get extended over time, resulting in potentially complex environments with effort-intensive maintenance requirements. Combined with the emergence of ever-growing, non-federated integration architectures that can be even more difficult to maintain and evolve, the average IT department can demand a significant amount of an organization's overall operational budget. For many organizations, the financial overhead required by IT is a primary concern because it often continues to rise without demonstrating any corresponding increase in business value.

Service-oriented computing advocates the creation of agnostic solution logic—logic that is agnostic to any one purpose and therefore useful for multiple purposes. This multi-purpose or reusable logic fully leverages the intrinsically interoperable nature of services. Agnostic services have increased reuse potential that can be realized by allowing

them to be repeatedly assembled into different compositions. Any one agnostic service can therefore find itself being repurposed numerous times to automate different business processes as part of different service-oriented solutions.

With this benefit in mind, additional up-front expense and effort is invested into every piece of solution logic so as to position it as an IT asset for the purpose of repeatable, long-term financial returns. As shown in Figure 3.31, the emphasis on increasing ROI typically goes beyond the returns traditionally sought as part of past reuse initiatives. This has much to do with the fact that service-orientation aims to establish reuse as a common, secondary characteristic within most services.

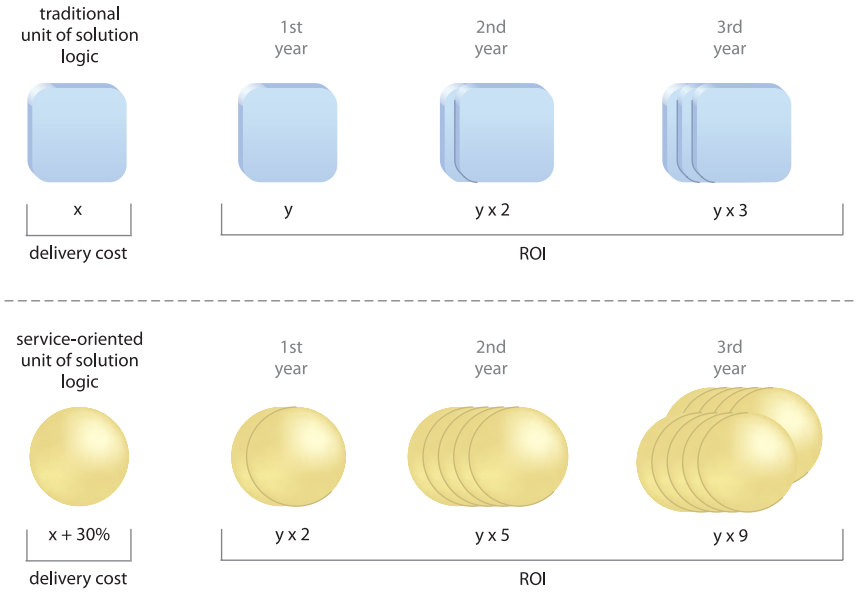


Figure 3.31
An example of the types of formulas being used to calculate ROI for SOA projects. More is invested in the initial delivery with the goal of benefiting from increased subsequent reuse.

It is important to acknowledge that this goal is not simply tied to the benefits traditionally associated with software reuse. Proven commercial product design techniques are incorporated and blended with existing enterprise application delivery approaches to form the basis of a distinct set of service-oriented analysis and design processes (as described earlier in the *Service-Oriented Analysis and Service Modeling* and *Service-Oriented Design* sections).

Increased Organizational Agility

Agility, on an organizational level, refers to the efficiency with which an organization can respond to change. Increasing organizational agility is very attractive to corporations, especially those in the private sector. Being able to more quickly adapt to industry changes and outmaneuver competitors has tremendous strategic significance.

An IT department can sometimes be perceived as a bottleneck, hampering desired responsiveness by requiring too much time or resources to fulfill new or changing business requirements. This is one of the reasons agile development methods have gained popularity as they provide a means of addressing immediate, tactical concerns more rapidly.

Service-oriented computing is very much geared toward establishing wide-spread organizational agility. When service-orientation is applied throughout an enterprise, it results in the creation of services that are highly standardized and reusable and therefore agnostic to parent business processes and specific application environments.

As a service inventory is comprised of more and more of these agnostic services, an increasing percentage of its overall solution logic belongs to no one application environment. Instead, because these services have been positioned as reusable IT assets, they can be repeatedly composed into different configurations. As a result, the time and effort required to automate new or changed business processes is correspondingly reduced because development projects can now be completed with significantly less custom development effort (Figure 3.32).

The net result of this fundamental shift in project delivery is heightened responsiveness and reduced time to market potential, all of which translates into increased organizational agility.

NOTE

Organizational agility represents a target state that organizations work toward as they deliver services and populate service inventories. The organization benefits from increased responsiveness *after* a significant amount of these services is in place. The processes required to model and design services require more up-front cost and effort than building the corresponding quantity of solution logic using traditional project delivery approaches.

It is therefore important to acknowledge that service-orientation has a strategic focus that intends to establish a highly agile enterprise. This is different from agile development approaches that have more of a tactical focus due to an emphasis on delivering solution logic more rapidly. From a delivery perspective, service-orientation does not tend to increase agility.

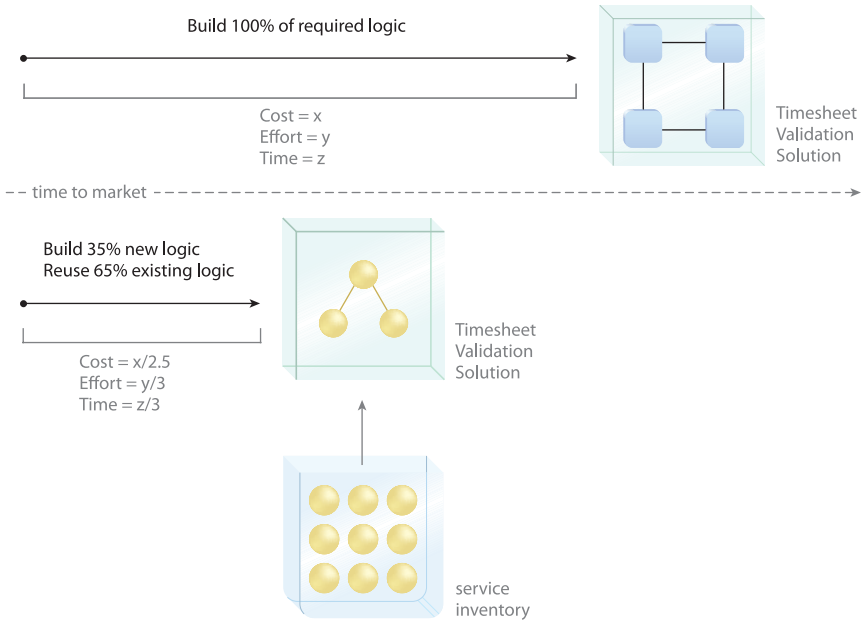


Figure 3.32
Another example of a formula used in SOA projects. This time, the delivery timeline is projected based on the percentage of "net new" solution logic that needs to be built. Though in this example only 35% of new logic is required, the timeline is reduced by around 50% because additional effort is still required to incorporate existing, reusable services from the inventory.

Reduced IT Burden

Consistently applying service-orientation results in an IT enterprise with reduced waste and redundancy, reduced size and operational cost (Figure 3.33), and reduced overhead associated with its governance and evolution. Such an enterprise can benefit an organization through dramatic increases in efficiency and cost-effectiveness.

In essence, the attainment of the previously described goals can create a leaner, more agile IT department; one that is less of a burden on the organization and more of an enabling contributor to its strategic goals.

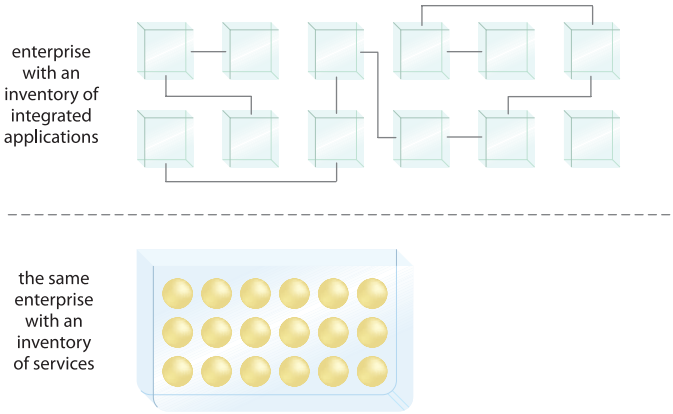


Figure 3.33
If you were to take a typical automated enterprise and redevelop it entirely with custom, normalized services, its overall size would shrink considerably, resulting in a reduced operational scope.

SUMMARY OF KEY POINTS

- Key benefits of service-oriented computing are associated with the standardization, consistency, reliability, and scalability established within services through the application of service-orientation design principles.
- The service-oriented computing platform provides the potential to elevate the responsiveness and cost-effectiveness of IT through a design paradigm that emphasizes the realization of strategic goals and benefits.

3.4 CASE STUDY BACKGROUND

The Cutit ownership team has nowhere near the resources or in-house expertise to plan a transition toward an SOA-based automation environment. They therefore engage a local consulting firm to take charge of the planning and analysis effort. The goal is to complete this project within a month and then use the resulting reports to decide on a delivery strategy.

The consultants spend the next few weeks invading Cutit's environments to document technology and business requirements. They look at service encapsulation options for legacy systems and service-based middleware platforms as part of a marketplace survey but also perform some analysis around the creation of custom services to replace the outdated automation hub.

As part of the final analysis, a preliminary service-oriented architecture is conceptualized and supplemented with a list of Web service-centric technology components required to establish it. Cutit reviews the reports and takes the consultants' recommendations into consideration. The report emphasizes the pursuit of reuse, but Cutit is more interested in leveraging service-oriented computing to establish unity across its modest enterprise and to achieve a state where solution logic can be more easily extended in response to unpredictable business demands.

Regardless, Cutit decides to proceed to the next step. Before moving ahead and building actual services, they invest in the creation of a service inventory blueprint. Cutit cannot afford to wait more than three weeks before entering the development stage, so this model will need to be high-level and therefore somewhat incomplete.

About the Author

Thomas Erl is the world's top-selling SOA author, the Series Editor of the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*, and Editor of *The SOA Magazine*.

With over 65,000 copies in print, his first two books, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services* and *Service-Oriented Architecture: Concepts, Technology, and Design* have become international bestsellers and have been translated into several languages. Books by Thomas Erl have been formally reviewed and endorsed by senior members of major software organizations, including IBM, Sun, Microsoft, Oracle, BEA, HP, SAP, Google, and Intel.

Thomas is also the founder of SOA Systems Inc. (`www.soasystems.com`), a company specializing in SOA training and strategic consulting services with a vendor-agnostic focus. Through his work with standards organizations and independent research efforts, Thomas has made significant contributions to the SOA industry, most notably in the areas of service-orientation and SOA methodology.

Thomas is a speaker and instructor for private and public events, and has delivered many workshops and keynote speeches. For a current list of his workshops, seminars, and courses, see `www.soatraining.com`.

Papers and articles written by Thomas have been published in numerous industry trade magazines and Web sites, and he has delivered Webcasts and interviews for many publications, including the *Wall Street Journal*.

For more information, visit `www.thomaserl.com`.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services

ISBN 0131428985

This top-selling field guide offers expert advice for incorporating XML and Web services technologies within service-oriented integration architectures.

Service-Oriented Architecture: Concepts, Technology, and Design

ISBN 0131858580

Widely regarded as the definitive "how-to" guide for SOA, this best-selling book presents a comprehensive end-to-end tutorial that provides step-by-step instructions for modeling and designing service-oriented solutions from the ground up.

SOA: Principles of Service Design

ISBN 0132344823

Published with over 240 color illustrations, this hands-on guide contains practical, comprehensive, and in-depth coverage of service engineering techniques and the service-orientation design paradigm. Proven design principles are documented to help maximize the strategic benefit potential of SOA.

SOA: Design Patterns

ISBN 0136135161

Software design patterns have emerged as a powerful means of avoiding and overcoming common design problems and challenges. This new book presents a formal catalog of design patterns specifically for SOA and service-orientation. All patterns are documented using full-color illustrations and further supplemented with case study examples.

Several additional series titles are currently in development and will be released soon. For more information about any of the books in this series, visit www.soabooks.com.

PRENTICE HALL