The Definitive Guide to SOA: Oracle Service Bus, Second Edition

Copyright © 2008 by Jeff Davies, David Schorow, Samrat Ray, and David Rieber

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1057-3

ISBN-13 (electronic): 978-1-4302-1058-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Lead Editor: Steve Anglin
Technical Reviewer: Jay Kasi
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor/Artist: Kinetic Publishing Services, LLC
Proofreader: Nancy Sixsmith
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

Why Use a Service Bus?

Enterprise service buses (ESBs) are all the rage in modern software development. You can't pick up a trade magazine these days without some article on ESBs and how they make your life wonderful. If you're a software development veteran, you'll recognize the hype immediately. ESBs aren't going to be the magic answer for our industry any more than were XML, web services, application servers, or even ASCII. Each of the aforementioned technologies started life with a lot of fanfare and unrealistic expectations (the result of the inevitable ignorance we all have with any emerging technology), and each technology ended up becoming a reliable tool to solve a specific set of problems.

The same is true for the ESB. Putting the hype aside, let's focus on a bit of software history so we can better understand the problems that the ESB is designed to address.

The Problems We Face Today

Software development is a tough business. We expect modern software systems to have exponentially more functionality than we expected from them only a few years ago. We often develop these systems with ever-dwindling budgets and sharply reduced timeframes, all in an effort to improve efficiency and productivity. However, we cannot lament these issues. These very issues drive us to deliver software that's better, faster, and cheaper.

As we've raced to develop each generation of software system, we've added significantly to the complexity of our IT systems. Thirty years ago, an IT shop might have maintained a single significant software system. Today, most IT shops are responsible for dozens, and sometimes hundreds, of software systems. The interactions between these systems are increasingly complex. By placing a premium on delivering on time, we often sacrifice architecture and design, promising ourselves that we'll refactor the system some time in the future. We've developed technologies that can generate large quantities of code from software models or template code. Some of the side effects of this race into the future are a prevalence of point-to-point integrations between software applications, tight coupling at those integration points, a lot of code, and little configuration.

Point-to-Point Integrations

Software development today is tactical and project-oriented. Developers and architects frequently think in terms of individual software applications, and their designs and implementations directly reflect this thinking. As a result, individual applications are directly integrated with one another in a *point-to-point* manner.

A point-to-point integration is where one application depends on another specific application. For example, in Figure 1-1, the CustomerContactManager (CCM) application uses the BillingSystem interface. You can say that the CCM application "knows" about the BillingSystem application. You also hear this kind of relationship referred to as a *dependency*, because one application depends on another application to function correctly.

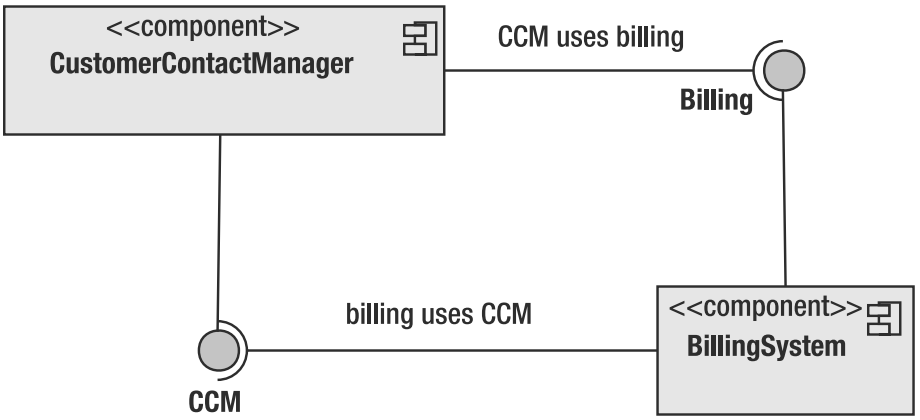

Figure 1-1. *Early point-to-point integrations*

Figure 1-1 illustrates a trivial IT environment, with only two applications and two point-to-point integrations. Just to be clear, the first integration allows the CCM system to call the BillingSystem application. The second integration point allows the BillingSystem application to call the CCM system. When your information technology (IT) department is this small, point-to-point integration is fairly easy to manage.

Figure 1-2 expands on the problem a bit. The IT shop is now home to 8 software systems and a total of 11 integration points. This illustrates a common pattern in integration: the number of integration points grows faster than the number of systems you're integrating!

Even Figure 1-2 is, by modern standards, a trivial IT system. A midsized service provider where Jeff once worked had 67 business systems and another 51 network systems—118 software systems integrated in a point-to-point manner is unmanageable. We know of telephone companies that have 12 or more billing systems. Having duplicates of certain software systems (such as billing) or having a large number of software systems in general is quite common; large companies can acquire smaller companies (and therefore acquire the software systems of the smaller companies) faster than most IT shops can integrate the newly acquired systems.

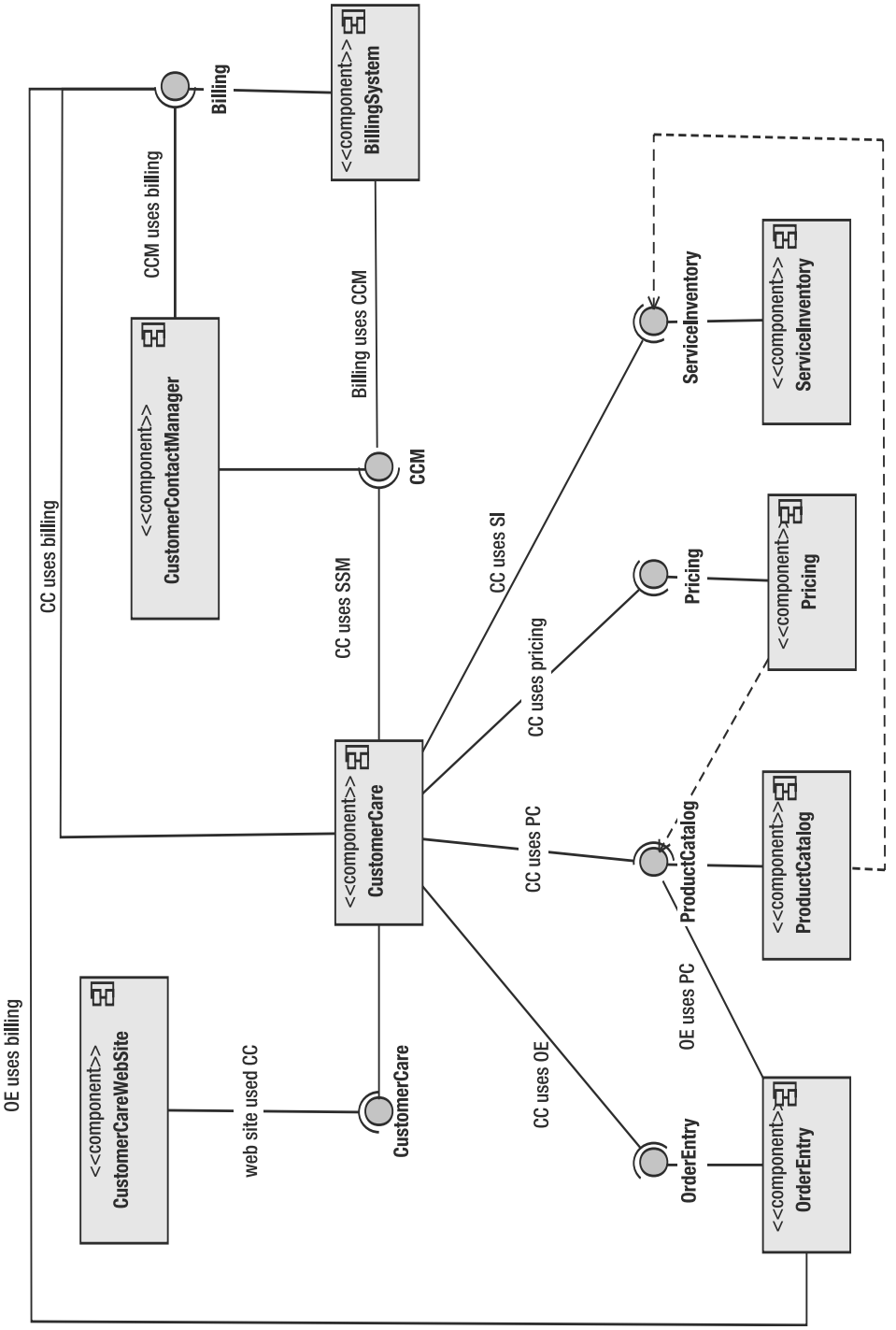

Figure 1-2. *Increasing point-to-point integration*

Tight Coupling

Tight coupling is often a by-product of point-to-point integrations, but it's certainly possible to develop tightly coupled applications no matter what your integration environment looks like. Loose coupling is desirable for good software engineering, but tight coupling can be necessary for maximum performance. Coupling is increased when the data exchanged between components becomes larger or more complex. In reality, coupling between systems can rarely be categorized as "tight" or "loose." There's a continuum between the two extremes.

Most systems use one another's Application Programming Interfaces (APIs) directly to integrate. For Enterprise JavaBeans (EJB) applications, you commonly create a client JAR file for each EJB application. The client JAR file contains the client stubs necessary for the client applications to call the EJB application. If you make a change to any of the APIs of the EJB application, you need to recompile and deploy the EJB application, recompile the client JAR, and then recompile and redeploy each of the client applications. Figure 1-3 illustrates this set of interdependencies among the software components and the file artifacts that realize them.

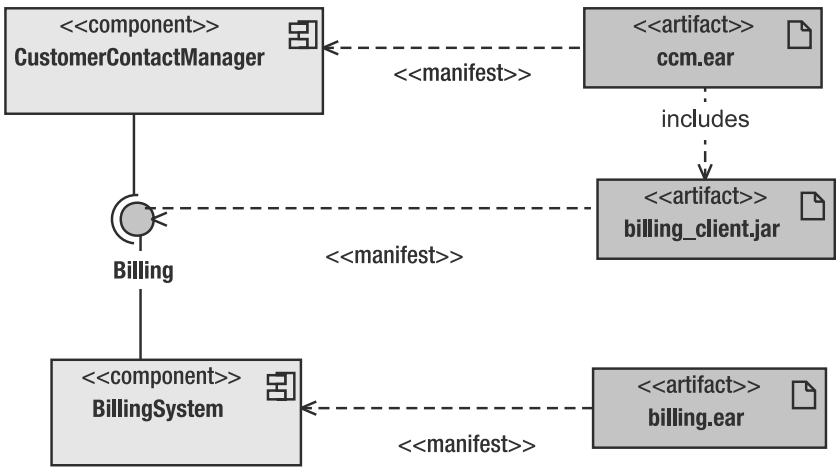

Figure 1-3. *EJB coupling model*

Tight coupling results in cascading changes. If you change the interface on which other components depend, you must then recompile the client applications, often modifying the client code significantly.

It's a common (and false) belief that you can use interfaces to reduce the coupling between systems. Interfaces are intended to abstract out the behavior of the classes that implement the interfaces. They do provide some loosening of the coupling between the client and the implementation, but their effect is almost negligible in today's systems. This is not to say that interfaces aren't useful; they most certainly are. But it's important to understand the reasons why they're useful. You still end up tightly coupled to a specific interface. Here is an example:

```
package com.alsb.foo;
public interface SampleIF {
   public int getResult(String arg1);
}
```

A client that depends on this interface is tightly coupled. If you change the `getResult()` method to take another argument, all clients of the interface must be recompiled. It's precisely this level of intolerance to change that tightly couples the code. The problem isn't so much in the design of the interface, but with the technology that implements the interface.

Enterprise Application Integration

Commonly referred to as EAI, enterprise application integration reached its peak in the 1990s. EAI now suffers the fate of many older technological approaches: being relegated to the category of "yesterday's" architecture. This reflects a bad habit we technologists have—any idea that seems smart today will be considered foolish or stupid tomorrow.

Contrary to popular belief, there is nothing wrong with EAI, and it remains a viable tool in our problem-solving tool chest. The only real problem with EAI is that it is misnamed; it is not really fit for enterprise-level architecture. EAI is fine for departmental-level integration or for simply deriving greater business value by integrating several software applications together so that they behave as a single meta-application.

The downside to EAI is twofold. First, EAI systems tend to employ a point-to-point approach to integrating applications. There is little or no abstraction of the component systems. This makes EAI systems just as brittle as any other point-to-point integration. The second flaw in the approach (from an enterprise perspective) is that all of the integration logic and any additional business logic are defined and maintained in the EAI tool, which lies at the center of the integration. EAIs are sometimes referred to as "spoke-and-hub" architecture because the EAI tool lies in the center of the integrated systems, like the hub of a great wheel, as illustrated in Figure 1-4.

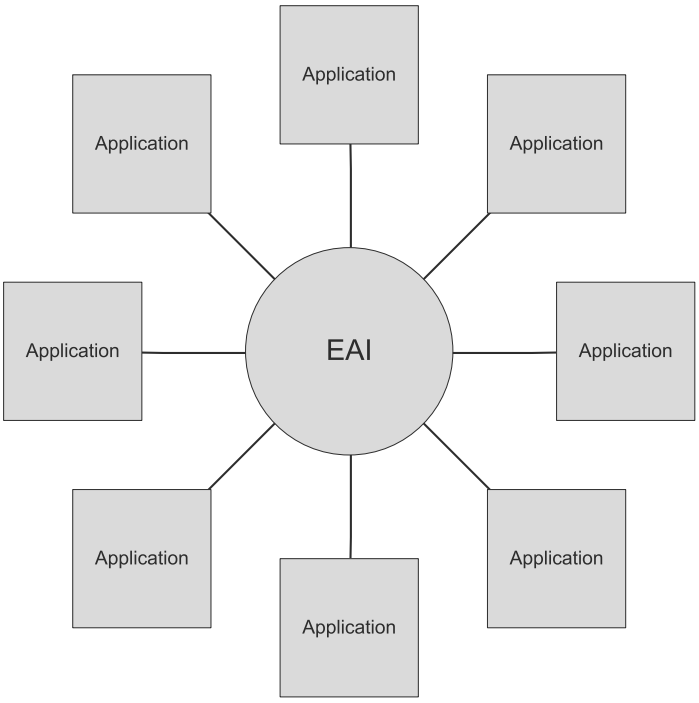

Figure 1-4. *EAI's "spoke-and-hub" architecture*

We are often asked how EAI differs from service-oriented architecture (SOA). The answer is simple. With EAI, all integration is done in the center—the hub. With SOA, the integration logic occurs at the edges of the architecture, not the heart. The integrations are pushed outward, toward the applications themselves, leaving the bus to "speak" a standardized language. Figure 1-5 illustrates the ESB/SOA architecture.

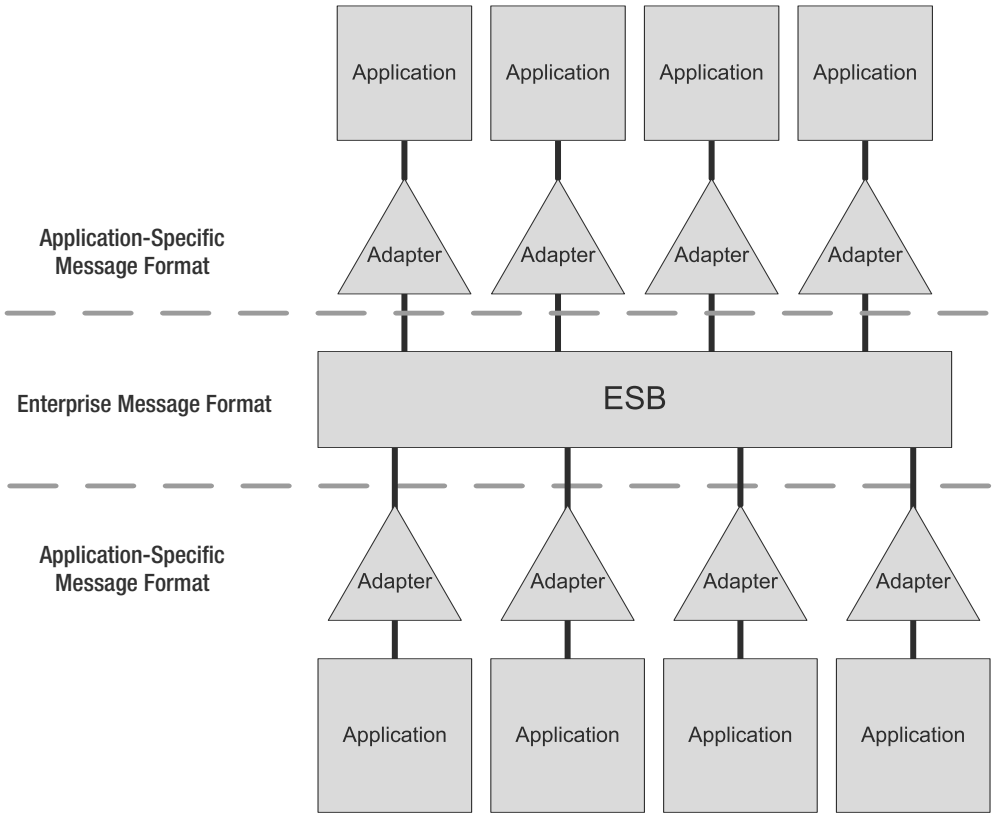

Figure 1-5. *ESB/SOA integrates at the edges.*

Every application "speaks its own language." By that, we mean that every application has its own way of representing information and working with that information. This language takes the form of an API. SOA recognizes that every enterprise also speaks its own language, which is why most applications are customized to meet the specific needs of the company. SOA uses an ESB to route and exchange messages at a high level of abstraction. Application-specific adapters are used to convert message formats from application-specific formats to enterprise-specific formats. SOA integrates at the edges of the applications, not in the heart of the enterprise.

This is not only an architectural approach, but also an architectural pattern. Enterprises have their own language, but so do the various departments within the enterprise. Each department has its own needs and way of doing business. Increasingly, we are seeing a hierarchical use of service buses to allow different departments, subsidiaries, or other business units to provide an abstraction layer between their applications and the interfaces they support. Figure 1-6 illustrates hierarchical ESB usage.

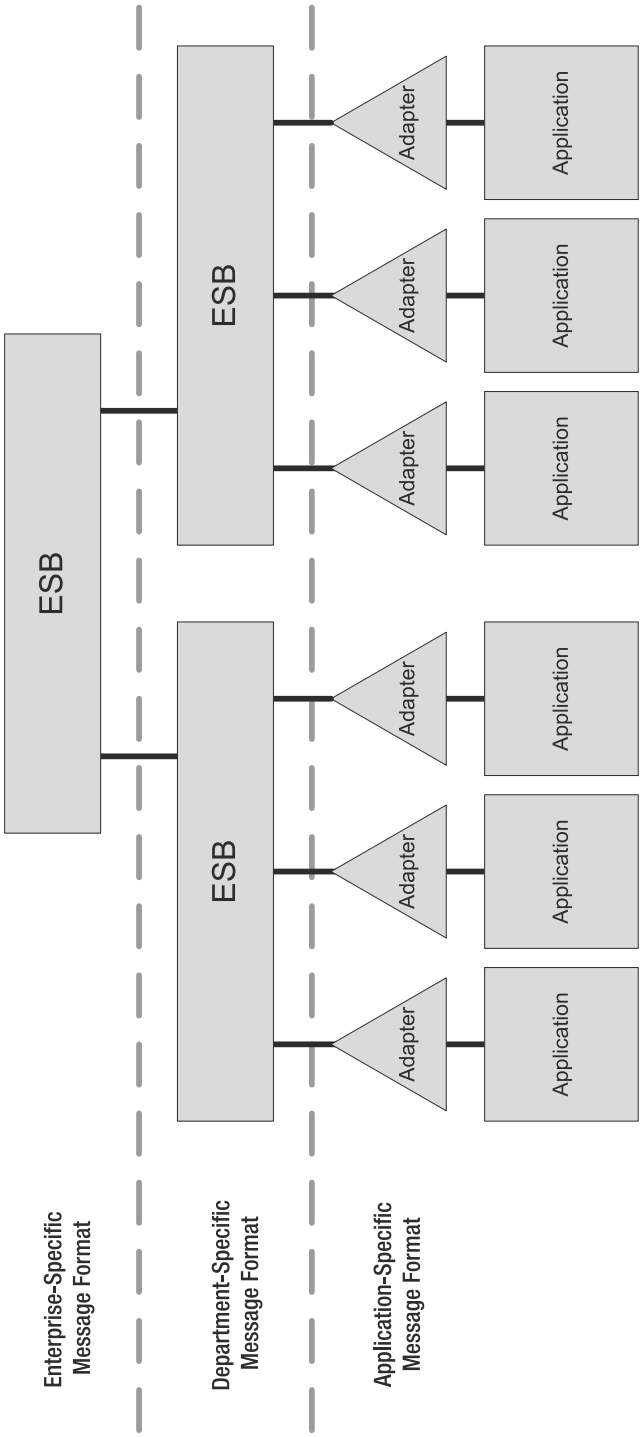

Figure 1-6. *Hierarchical ESB usage*

In Chapters 12 and 13, we will go into detail on the architectural principles used to achieve the specific goals of agility, resilience, and manageability, but the crux of the matter is hinted at in Figures 1-5 and 1-6. You must define formal layers of abstraction in order to have loose coupling and agility. Point-to-point integration approaches can never achieve these goals, no matter what technologies they may employ.

Early ESBs

Early ESBs were primarily concerned with making web services available to service consumers. Their implementation was clunky (as new technologies usually are) and didn't embrace many open standards, simply because those standards didn't exist at the time. Furthermore, the developers of early ESBs could only try to predict how web services would affect enterprise computing and IT organizations.

The early ESBs were "ESBs" in name only. As the industry has matured, so has our understanding of the role of an ESB in modern architecture. Today's ESBs must go far beyond simply "service-enabling" functionality. An ESB must also provide robust solutions for today's IT challenges.

Modern Solutions

The IT industry is constantly evolving. Our understanding of the issues that surround the management of large IT systems matures on a daily basis. Modern ESBs are simply the latest tools to help us manage our IT problems. They benefit from real-world examples of how SOA is changing the face of today's advanced corporations. Although early ESBs could only address a handful of the following issues, modern ESBs need to address them all.

Loose Coupling

You might have heard that web services provide you with loose coupling between systems. This is only partially true. Web services, by the very nature of Web Services Description Language (WSDL) and XML Schema Definition (XSD) document, can provide some loose coupling because they formalize a contract between the service consumer and the service provider. This is a "design-by-contract" model, and it does provide tangible benefits. If you're careful, you can create a schema that's platform-neutral and highly reusable.

However, if you take a look at any WSDL, you'll see that the service endpoints are written into the WSDL, as you can see in Listing 1-1.

Listing 1-1. *HelloWorld Service Definition*

```
<service name="HelloWorldService">
   <port binding="s1:HelloWorldServiceSoapBinding"
      name="HelloWorldPortSoapPort">
      <s2:address location="http://www.bea.com:7001/esb/Hello_World" />
   </port>
</service>
```

By specifying a specific machine and port (or a set of machines and ports), you're tightly coupling this service to its physical expression on a specific computer. You can use a Domain Name Server (DNS) to substitute portions of the URL, and therefore direct clients into multiple machines in a server farm. However, DNS servers are woefully inadequate for this, due to their inability to understand and manage the status of the services running on these servers.

So, loose coupling isn't achieved by WSDL or web services alone. A more robust solution is to provide some mediation layer between service clients and service producers. Such a mediation layer should also be capable of bridging transport, message formats, and security technologies. For example, a service might be invoked through a traditional HTTP transport mechanism, but it can then invoke lower-level services through Java Message Service (JMS), e-mail, File Transfer Protocol (FTP), and so on. This approach is often effectively used to "wrap" older services and their transports from the newer service clients.

Location Transparency

Location transparency is a strategy to hide the physical locations of service endpoints from the service clients. Ideally, a service client should have to know about a single, logical machine and port name for each service. The client shouldn't know the actual service endpoints. This allows for greater flexibility when managing your services. You can add, move, and remove service endpoints as needed, without needing to recompile your service clients.

Mediation

An ESB is an intermediary layer, residing between the service client and the service providers. This layer provides a great place for adding value to the architecture without changing the applications on either end. An ESB is a service provider to the service clients. When clients use a service on the service bus, the service bus has the ability to perform multiple operations: it can transform the data or the schema of the messages it sends and receives, and it can intelligently route messages to various service endpoints, depending on the content of those messages.

Schema Transformation

The web service published by the service bus might use a different schema from the schema of the business service it represents. This is a vital capability, especially when used in conjunction with a canonical taxonomy or when aggregating or orchestrating other web services. It's quite common that a service client will need to receive its data using a schema that's significantly different from that of the service provider. The ability to transform data from one schema to another is critical for the success of any ESB.

Service Aggregation

The service bus can act as a façade and make a series of web service calls appear as a single service. Service aggregation follows this pattern, making multiple web service calls on behalf of the proxy service and returning a single result. Service orchestration is similar to service aggregation, but includes some conditional logic that defines which of the lower-level web services are called and the order in which they are invoked.

Load Balancing

Due to their position in any architecture, ESBs are well suited to perform load balancing of service requests across multiple service endpoints. When you register a business web service with Oracle Service Bus (OSB), you can specify the list of service endpoints where that business service is running. You can change this list, adding or removing service endpoints without needing to restart the OSB server.

Enforcing Security

You should enforce security in a centralized manner whenever possible. This allows for a greater level of standardization and control of security issues. Furthermore, security is best enforced through a policy-driven framework. Using security policies means that the creation and application of security standards happen outside the creation of the individual web services.

Monitoring

An ESB plays a vital role in an SOA. As such, you must have a robust way to monitor the status of your ESB, in both proactive and reactive manners. The ability to proactively view the performance of the service bus allows you to help performance-tune the service bus for better performance. Tracking the performance over time can help you plan for increasing the capacity of your ESB.

Reactive monitoring allows you to define alerts for specific conditions. For example, if a specific service doesn't complete within a given timeframe, the ESB should be able to send an alert so that a technician can investigate the problem.

Configuration vs. Coding

A modern service bus should be configuration-based, not code-based. For many engineers, the importance of that statement isn't immediately obvious. It took us some time before we appreciated the configuration-oriented capability of OSB. Most software systems in use today are code-based. Java EE 5 applications are a great example of this. In a Java EE 5 application, you write source code, compile it into an EAR or WAR file, copy that EAR or WAR file onto one or more Java EE 5 application servers, and then deploy those applications. Sometimes it's necessary to restart the Java server, depending on the nature of your deployment.

Configuration-based systems work differently. There's nothing to compile or deploy. You simply change the configuration and activate those changes. We would argue that your telephone is configuration-based; you configure the telephone number you want to call, and your call is placed. There's no need to restart your phone. Similarly, network routers and switches are configuration-based. As you make changes to their configuration, those changes take effect. There's no need for a longer software development life cycle to take place.

Configuration and coding are two different strategies. Neither is superior to the other in all situations. There are times when the Java EE 5 approach is the most appropriate, and other times when the configuration-based approach is best.

Enter Oracle Service Bus

BEA released AquaLogic Service Bus in June 2005. With the Oracle acquisition of BEA in mid-2008, the product was rebranded to Oracle Service Bus (OSB). OSB runs on Windows, Linux, and Solaris platforms. OSB is a fully modern ESB and provides functionality for each of the capabilities expected from today's enterprises, as described in the following sections.

Loose Coupling

Aside from the loose coupling benefits from WSDL and XSD, OSB adds the ability to store WSDL, XSD, eXtensible Stylesheet Language Transformation (XSLT), and other information types within the OSB server as "resources." These resources are then made available throughout the OSB cluster of servers, allowing you to reuse these resources as needed.

The benefit of this might not be immediately clear, so we'll give an example. Many companies define and manage enterprise-wide data types using an XML document schema. Because OSB can store an XML document schema as a resource in the service bus, that schema can easily be reused by any number of WSDLs or other XSDs. This enables you to create and enforce enterprise-wide standards for your data types and message formats.

Location Transparency

One of the capabilities of OSB is to register and manage the locations of various web services within the enterprise. This provides a layer of abstraction between the service client and the service provider, and improves the operational aspect of adding or removing service providers without impact to the service clients.

Mediation

One of the roles for which OSB is specifically designed is that of a service mediator. OSB uses the paradigm of "proxy services" and "business services," where the proxy service is the service that OSB publishes to its service clients, and the business services are external to OSB. In between the proxy service and the business service is the layer where service mediation takes place. Schemas can be transformed, as can the data carried by those schemas. Intelligent or content-based routing also takes place in this mediation layer.

Schema Transformation

Schema transformation is a central capability of OSB. OSB provides a number of ways to transform schemas, depending on your specific needs. You can use XSLT to transform XML data from one schema to another. Similarly, you can use XQuery and XPath to perform XML document schema transformations. Additionally, OSB supports the use of Message Format Language (MFL) to format schemas to and from non-XML formats, such as comma-separated value (CSV) files, COBOL copy books, Electronic Data Interchange (EDI) documents, and so on.

Service Aggregation

OSB doesn't match a single proxy service to a single business service. Instead, OSB allows you to define a many-to-many relationship between proxy services and business services. This approach allows for service aggregation, orchestration, and information enrichment.

Load Balancing

Because OSB registers the service endpoints of all business services, it's ideally situated for operating as a load balancer. This is especially true because OSB is configuration-based, not code-based. As a result, you can add or remove service endpoints from a business service and activate those changes without needing to restart your service bus.

Enforcing Security

OSB, as a service mediator, is ideally situated to enforce the security of the web services because it operates on the perimeters of the enterprise. OSB is designed to enforce security through the use of explicit security policies. Using OSB, you can propagate identities, mediate, and transform between different security technologies, such as Basic Authentication, Secure Sockets Layer (SSL), and Security Assertion Markup Language (SAML).

Monitoring

OSB provides a robust set of features around monitoring. The service bus console allows you to look proactively at the state of your entire ESB.

For reactive monitoring, OSB allows you to define alerts for conditions that you define. Alerts can be delivered via e-mail to specified recipients. We'll discuss monitoring more fully in Chapter 10.

Configuration vs. Coding

OSB is a configuration-based service bus. You don't write Java code for OSB, although OSB can recognize and make use of Java code in some circumstances. Instead, you configure OSB through its web-based console.

One handy feature of the OSB console is that your configuration changes don't take effect when you make each change. Instead, your configuration changes are grouped together, similarly to a database transaction, and take effect only when you tell OSB to activate your changes. This is a critical capability, because many times you'll make multiple changes that are interdependent.

Of course, creating these changes by hand can be an error-prone process. To avoid mistakes, OSB allows you to make changes in one environment (a development or a test environment), and then export those changes as a JAR file. You can then import that JAR file into your production environment as a set of configuration changes, and further customize the configuration file for a specific deployment environment upon import. This process allows you to script your changes and activate them as if you had entered those changes directly into the OSB console by hand.

Won't This Lock Me into Oracle Technologies?

OSB is entirely standards-based. You configure OSB through the use of XQuery, XPath, XSLT, and WSDLs. The only aspect of OSB that might be deemed "proprietary" is the implementation of the message flows (see Chapter 4). However, these message flows are simply graphical constructs for common programming logic, and they're easy to reproduce in just about any programming language. The real heavy lifting in OSB is done using the open standards for functionality, and WebLogic Server for reliability and scalability.

Because OSB is standards-based, it's designed to integrate with and operate in a heterogeneous architecture. Using OSB as a service bus doesn't preclude you from using other technologies in any way. OSB is used to integrate with .NET applications, TIBCO, SAP, JBoss, WebSphere, Siebel, and many more technologies. Oracle didn't achieve this level of heterogeneity by accident; it's all part of the company's "blended" strategy, which involves using open standards and open source to achieve the maximum amount of interoperability.

Why Buy an Enterprise Service Bus?

We come across this question frequently. The truth is that an ESB contains no magic in it at all. It's possible to build your own ESB from scratch. In fact, one of the authors has done it twice before joining Oracle. There's nothing that the engineers at Oracle can write that you cannot write yourself, given enough time, money, and training. This principle holds true for all software. You don't need to use Microsoft Word to write your documents; you could create your own word processor. In the same way, HTML standards are publicly available, and you could use your engineering time to develop your own web browser.

Naturally, few of us would ever consider writing our own word processor or web browser. It's a far better use of our time and money either to buy the software or to use an open source version. This is especially true if your company isn't a software company. If you work in an IT shop for a company whose primary line of business isn't software, you'll recognize the fact that building software from scratch is a difficult sell to your executive staff. There simply is no return on investment for such development efforts. Your time and skills are better spent solving problems specific to your company.

There are a number of benefits to purchasing OSB. First is the fact that it comes from a dyed-in-the-wool software company. Oracle has been in business for more than three decades and has a long history of delivering innovative, successful products. Furthermore, Oracle supports those products for many years.

A number of open source ESBs are available today. Most are in the early stages of development and functionality. Although we love open source and advocate its use in many areas, we would be hesitant to use an open source ESB. An ESB will become the central nervous system of your enterprise. You should exercise caution and diligence when selecting an ESB. You want one with a proven record of success, from an organization that works hard to keep itself ahead of current market demands.

OSB is built on Oracle's WebLogic Server technology. This gives you enterprise-quality reliability and scalability. On top of this, OSB is built on open standards for maximum interoperability in a heterogeneous environment. It's an ESB that will carry your company into the future.

Summary

In this chapter, we reviewed the features and functions that a modern ESB should have, and we've described each feature's importance to the organization. OSB implements all these features, and it possesses many more advanced features that we'll cover in this book. But we've talked enough about OSB. It's time to start to demonstrate, in working code, exactly how to use these features to their fullest.