EVENT-DRIVEN ARCHITECTURE

How SOA Enables the Real-time Enterprise

Hugh Taylor | Angela Yochem | Les Phillips | Frank Martinez

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Event-driven architecture : how SOA enables the real-time enterprise / Hugh Taylor ... [et al.].

p. cm.

Includes bibliographical references.

ISBN-13: 978-0-321-32211-1 (pbk. : alk. paper)

ISBN-10: 0-321-32211-8 (pbk. : alk. paper) 1. Service-oriented architecture (Computer science) 2. Discrete-time systems. 3. Business--Data processing. 4. Business enterprises--Computer networks. I. Taylor, Hugh.

TK5105.5828.E94 2009

004.6'82--dc22

2008055032

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-32211-1
ISBN-10: 0-321-32211-8

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing March 2009

Editor-in-Chief
Mark Taub

Acquisitions Editor
Greg Doench

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Betsy Harris

Copy Editor
Karen Annett

Indexer
Ken Johnson

Proofreader
Debbie Williams

Technical Reviewers
Cliff Berg
Kevin Davis
David Kane

Publishing Coordinator
Michelle Housley

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

Foreword

It's been 15 years since the dawn of the Web, and we are still absorbing the lessons it teaches us about decentralization, loose coupling, standards, and resource representation. Even when technology seems to move quickly, it can take a long time to understand, appreciate, and apply the core principles it embodies.

The roots of event-driven architecture run even deeper. Twenty-five years ago, the graphical user interface forever changed how we think about applications. Suddenly, the event loop became a central organizing principle. Programs listened to events, processed them, and responded to them—sometimes by firing new events. There was no other way to effectively support fickle and unpredictable users who are liable to do anything whenever they want.

Enterprise software systems, of course, serve whole populations of fickle and unpredictable users. Some are customers, some are suppliers, and some are business partners. Here too, effective software has to listen well and respond intelligently.

It sounds simple, and conceptually it is. But while the stream of events produced by a GUI application is defined by the operating system, and is well understood by the programmer, an enterprise application lives in a connected world. Just as the resource-oriented Web architect has to learn how to design stateless resources, so must the event-oriented enterprise architect learn how to design stateless events.

But if EDA presents new challenges, it also emerges in an era of new opportunity. The tools and techniques of service-oriented architecture are becoming more mature, more interoperable, and more manageable.

With a strong SOA skeleton in place, EDA can weave the enterprise's nervous system. This book explains why event-driven architecture yields smart and resilient enterprise software, and shows you how to start "thinking EDA."

—Jon Udell

Preface

About This Book

As professionals in the enterprise architecture field, we have observed the recent and spectacular rise of the concept of service-oriented architecture (SOA) with excitement tempered by concern. The new standards-based architectural paradigm promises great advances in interoperability among previously incompatible software applications. In turn, it has the potential to deliver gains in agility and IT cost control. Perhaps most exciting, though, is the potential for SOA to make possible the realization of event-driven architecture (EDA), an approach to enterprise architecture that yields a high level of agility by increasing systems' awareness and intelligent responses to relevant events.

At the same time, it became clear to us that the steps required to design and deploy an EDA, or an SOA, its master set of architectural characteristics, were far from obvious. Even going beyond the fact that the technology and standards are immature and, thus, challenging, the practice of uniting software with an overarching standards-based approach that extends outside the enterprise is a new field, lacking in many of the guiding principles of infrastructure, governance, and best practices that hold together most traditional forms of architecture and development.

For better or worse, some software vendors are now bringing what they call EDA suites to market. However, the commercial offerings in EDA tend to be quite narrowly defined and vendor-centric. As such, they are inadequate on their own to offer much in the way of instruction on the overall best practices required for EDA.

We perceive a need among architects for a book that combines both the theory of EDA—the grand vision that led to its formation and the

essential nature of the paradigm—with a practical look at building an EDA over an SOA implementation in the real world. This book is neither all theory nor all practice. It is a blend, with the idea that true success with EDA depends on a good understanding of both aspects of the paradigm.

To understand how this book is set up and what it contains, we thought it would make sense first to take a quick look at the definition, history, and context of EDA and SOA. These two related architectural styles are not as new as they seem, though recent developments in standards have led to breakthroughs in their potential realization.

Inside This Book: The Path to EDA

Even for a lot of experienced architects and developers, the implicit connection between EDA and SOA has not been intuitively obvious. A lot of IT pros react to SOA with a sentiment akin to, "That's really cool. Now what?" These questions are completely legitimate. Imagine someone handing you a violin and declaring, "Oh, good, now I get to hear Mozart." That person is making several assumptions, including that you know what the violin is, how to play it, and how to play Mozart in particular. In many IT situations, it is not always evident how loose coupling and a service orientation will take you to an EDA. If your boss drops a Visual Studio 2008 pack on your desk and says, "Now you will deliver an EDA," you might not necessarily know how to get from here to there. That is the purpose of this book.

Much of this book is dedicated to helping you understand where the rubber meets the road in turning the vision of EDA into a reality. In so doing, we delve into detail on the subject of SOA, providing the essential building blocks of the most versatile and effective EDAs. We address one of the great unanswered questions posed in the wake of SOA's high-profile arrival on the IT scene: How do you actually get to the achievement of business goals that EDA enables using the actual technologies that make up SOA? The leap from Web services and SOA to the fulfillment of EDA, and its attendant agility and IT cost savings, requires some serious discipline.

Part I—The Theory of EDA

This book consists of two parts. Part I, "The Theory of EDA," covers the theoretical aspects of EDA. The path to EDA, which we guide you through in this book, starts with an understanding of what EDA is. Part I begins with a thorough theoretical definition of EDA. We cover the core components of EDA, such as event consumers and producers, message backbones, Web service transport, and so on. We also describe the basic patterns of EDA, including simple event processing, event stream processing, and complex event processing.

From this definition, we then explore the current context of EDA, which is the jungle of interoperability challenges that we all face in large enterprises. Having thus set up the situation that we face—we want EDA (or at least, we should consider it)—we see how difficult it can be to attain. Enter SOA, and its open interoperability, which paves the way for the realization of EDA.

In addition to defining EDA, we explore the SOA-EDA connection in depth. In our view, any serious attempt to develop an EDA today will rely on the use of SOA technology as it is emerging in the marketplace. The EDA of tomorrow will run on Web services and enterprise service buses. The EDA components—the event producers, consumers, and processors—will all be Web services. We will flesh out this vision of EDA.

The conclusion of Part I consists of examples of EDAs and how they might function. We explore examples of how businesses and other organizations might ideally use EDA to further their objectives. This set of examples provides a transition to Part II, "EDA in Practice," of the book, which moves you into the reality of EDA and how it might be approached in an actual enterprise setting.

Part II—EDA in Practice

Part II begins with Chapter 6, "Thinking EDA." This chapter explores ways to identify the ideal use of an EDA, or a partial EDA in realizing a set of business objectives. Chapters 7 and beyond present a set of case studies of EDA. Some of these case studies are based on real companies. Others are partially hypothetical, but based on real-life experiences we have had in the world of enterprise architecture.

In each case study, we describe the organizations involved as well as the technological and business challenges and objectives that they have. We look at the ways in which the business and technological situation would benefit from an EDA approach. We look at the practical issues

that arise in its design and implementation. Our goal is to include, where relevant, some organization and non-IT issues, such as project management and communication. Of course, we get into depth on the technologies required to birth the EDA.

Throughout the case studies, we look at a number of related topics in the field of enterprise architecture that have relevance for learning about EDA. These include SOA infrastructure, governance, and security. Wherever possible, we try to point out business issues that are relevant, but perhaps not apparent to the technology reader, as well as technology issues that might not be noticed by the business reader.

One of our other goals is to instill in you a good sense of when to use an EDA approach and when not to, for the paradigm is not a panacea for all IT and business problems. This issue reminds of the story of a man who once approached a famous surgeon and said, "You make more money in a week than I make in a year. I don't think it's fair. Is what you do so special?" The surgeon replied, "Surgery itself isn't that complicated. I could probably teach you to do it in a few weeks. What takes the training and skill is knowing when not to operate, and what to do when something goes wrong. Learning those two things can take years."

So it is with EDA. Developing a Web service is not hard for an experienced developer. Knowing how to use the functions of an SOA to create an EDA, though, is another matter. And, like the surgeon, you would be well served by understanding when to use and not use the EDA approach. If you take away nothing else from this book, consider that there are many cases where an EDA is not the optimal solution to a business issue.

Who Should Read This Book, and How They Should Read It

If you're holding this book in your hand, you are probably involved with information technology. If you are not in technology, we really admire your desire to be a broadly informed citizen. We have written this book in fairly deep, but not excessively detailed, technical language.

This is not a book that is awash in code or extensive jargon. We have made the choice to skip the deep, deep techie language because of the likely blend of readers that we expect to find. The subject of EDA can be of interest to the work of a vast audience. EDA itself is an area that is

inherently interdisciplinary. EDA naturally throws together developers, line-of-business people, IT managers, security specialists, architects, and network operations people. There is probably a whole EDA book for each of those disciplines. Luckily for us, someone else will write them. We want to present the topic in a unified approach that a multiplicity of readers can absorb.

Our other guess is that you probably work at a large organization or with an entity that interfaces with large organizations. Whether you work at a corporation, public sector organization, or educational institution, the issues for EDA are the same. We come from the corporate world, so we have a tendency to talk about "business value" a lot. If you can't relate to this, we are sorry, but for stylistic reasons we need to use just one measure of efficiency, and in our world, that measure is usually dollars. So, when we talk about "business value," we mean the economy of effort required to produce a result. It's a concept that can translate into any organizational agenda.

Introduction

Event-Driven Architecture: A Working Definition

Event-driven architecture (EDA) falls into the maddening category of a technology paradigm that is half understood by many people who claim to know everything about it. Although we recognize that we, too, might not know absolutely everything there is to know about EDA, we believe that it is necessary to set out a working definition of EDA that we can adhere to throughout this book. Getting to an effective working definition of EDA is challenging because EDA must be described at a sufficiently high level that is comprehensible to nontechnologists, but at the same time not so high level as to sound vague or irrelevant.

An event-driven architecture is one that has the ability to detect events and react intelligently to them. For our purposes—and we discuss this in great detail later on—an *event* is a change in *state* that merits attention from systems. Brenda Michelson, a technology analyst, writes, "In an event-driven architecture, a notable thing happens inside or outside your business, which disseminates immediately to all interested parties (human or automated). The interested parties evaluate the event, and optionally take action."[1]

One of the simplest examples of an event-driven system is actually from the noncomputer world. It is known as a thermostat. The thermostat is a mechanical device that turns the heat on or off based on its programmed reaction to an event, which is a change in temperature. The shift in temperature is the event, the "change in state" that triggers the reaction of the thermostat, which, in turn, affects the action of the heater.

We can see another simple example in the evolution of the automobile. Cars are becoming increasingly intelligent by reacting intelligently to their surroundings. If rain hits the windshield, the automobile recognizes the rain event and automatically turns on the windshield wipers,

turns on the headlights, and adjusts the front windshield defroster. All of these things were formerly the driver's responsibility, but now the car's internal system uses its intelligence to react. An EDA is an architecture that acts in the same way: It detects events and reacts to them in an intelligent way. To be able to detect events and react to them intelligently, an EDA must have certain capabilities, including the ability to detect events, transmit messages among its components that an event has occurred, process the reaction to the event, and initiate the reaction to the event if that is called for. In generic architectural terms, these capabilities translate into the concepts of *event producers, event consumers, messaging backbones,* and *event processors.* These go by many different names in practice, and this is one of the great hurdles to getting a feel for what an EDA is at its core.

Many examples of EDAs occur in the realm of information systems, though most of the ones currently deployed are limited in scope. For example, if your credit card is simultaneously used in two separate geographical locations, those two events can be "heard" by the credit card processing systems and examined for a potential fraud pattern. The credit card fraud detection EDA is set up to listen for events that indicate potential fraud and respond—or not respond—depending on a set of rules that are programmed into the event processors. If the charge occurring out of state is at a mail order merchant where you have shopped before, the system might not deem the event pattern to be a fraud. If the second charge is for a high-dollar value at a merchant where you have not shopped before, the EDA might trigger a response that places a warning or "watch" status on your credit card account. Or, the activity might prompt a person to call you and find out if you have lost your card.

Or, imagine that an FAA air traffic control application needs to know the probability of rain in a certain location. At the same time, the Air Force needs the same data, as does NASA. Assuming that the weather data is collected and available on a server somewhere, it is possible to tightly couple that server, and the software running on it, with the FAA, Air Force, and NASA's respective systems. Although this type of approach is frequently used, it is far easier to arrange for the weather application to *publish* the weather data and enable the *subscribers* (the FAA et al.) to get the data they need and use it however they need to use it. The *weather status event* of the weather application publishes the weather data so that the data subscribers can use it to drive the architecture. This is an event-driven architecture.

In this EDA, the FAA, Air Force, and NASA are integrated with the weather system by virtue of the fact that there is no specific coupling between the applications. Of course, they exchange data, but the applications are completely separate and have no inherent knowledge of one another. The developers do not need to know each other, and there is no need to coordinate. However, for it to work, they do need standards. To effectively disseminate and process events, the publisher and the subscriber might agree to use a commonly understood message format and a compatible transport mechanism.

One commonly used technology that is analogous to EDA is the Web itself. When you use a browser, you are initiating an integrated session with a remote system of which you have no specific knowledge. In all probability, you have no idea who programmed it, what language it's written in, where it is, and so on. Yet, your browser can pull whatever information it is permitted to get and show it to you in a format that you can understand. The event of requesting the uniform resource locator (URL) triggers the action that results in the display of the Hypertext Markup Language (HTML) content in your browser window. As we develop our explanation of EDA, though, you will see that the Web is a very simple EDA.

An EDA consists of applications that are programmed to publish, subscribe, or take other actions upon events triggered by applications with which they share no formal coupling. For this reason, EDA has been likened to a "nervous system" for the enterprise.

The Enterprise Nervous System

Where would the IT industry be without metaphors? Even the idea of using the word *architecture* to describe how Byzantine networks of hardware, software, and data are configured shows how reliant we are on abstraction to achieve an understanding of what we are trying to accomplish in enterprise IT. In the spirit of metaphors, then, we shall borrow a concept from human physiology, the central nervous system, to further our understanding of EDA.

If your cat steps on your toe, how do you know it? How do you know it's a cat, and not a lion? You might want to pet the cat, but shoot the lion. When the cat's paw presses against your toe, the nerve cells in your toe fire off a signal to your brain saying, "Hey, someone stepped on my toe." Also, they send a message that says something like, "It doesn't hurt that much" and "It was probably a cat." Or, if you saw the cat, the signals from

your optic nerve are synthesized with those from your toe, each invoking your mental data store of animals and likely toe steppers, and you should know pretty quickly that it was, indeed, a cat that stepped on your toe. Your central nervous system is a massively complex set of sensory receptors, wires, and integration points, known as synapses. The nervous system is critical to your functioning and survival in the world. Just imagine if your central nervous system didn't work well and you confused the cat with the lion. As Figure I.1 shows, you might shoot your cat and pet the lion, which would then eat you.

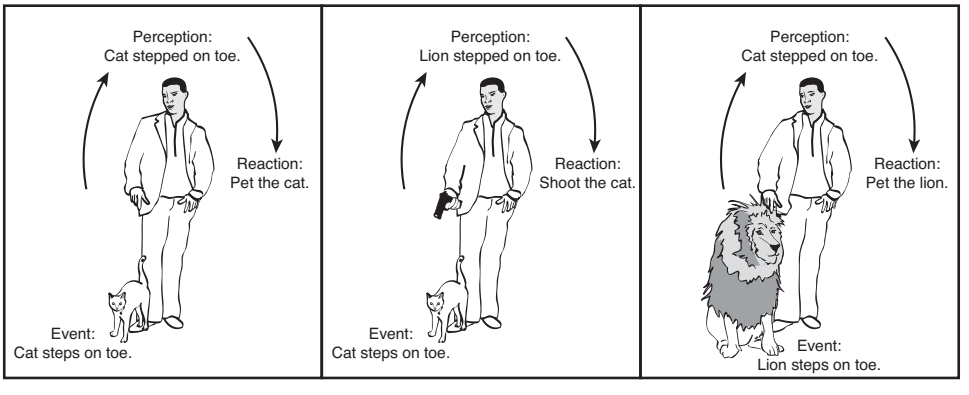

Figure I.1 The human nervous system as a metaphor for the enterprise. When the cat steps on your toe, do you recognize it as a cat, or mistake it for a lion?

Our enterprises have their own nervous systems, too. Our Web sites, enterprise resource planning (ERP) systems, customer relationship management (CRM) systems, databases, and network infrastructures, for example, all work to feed the corporate equivalent of sensory information to the corporate "brain." The corporate brain, in turn, assesses the input and reacts. Of course, the corporate brain might contain a few actual brains as well, in the form of employees, but their sensory input is determined by the enterprise nervous system. For example, if there is an increase in cash withdrawals at a bank, the banking systems, acting like nerve sensors in our toe, fire off withdrawal data to the corporate brain. The neurons in the corporate brain then route the data to its destination, which could be an automated bank cash reserve management system, the executive management team of the bank, or a combination. As our

brain assesses and reacts to the cat stepping on our toe, the corporate brain of the bank must assess the input of the withdrawal spike and react.

If our enterprises were living beings, most of them would need some pretty intensive neurological care. Unlike a well-functioning person, whose nervous system can learn how to react to different stimuli and determine the best way to handle a given situation based on sensory input and mental processing, the typical enterprise has a nervous system that is hardwired to react in a specific set of ways that might not be ideal for every situation. In the cat-on-toe situation, most of our enterprises would probably expect the worst and then shoot in the general direction of the cat. Or, perhaps a more realistic version of the metaphor—the enterprise wouldn't even know that anything had stepped on its toe, or that it even had a toe. It would be completely unaware most likely because it was never given the ability to be aware.

Like the person whose knee-jerk reaction is to shoot the cat regardless of what is going on, most of our enterprises have a nervous system that is not well set up to receive the data equivalent of sensory input, process it, know what it is, and react in an appropriate way. Event-driven architecture is an approach to IT that gives the enterprise the ability to improve its nervous system and have a level of adaptability and awareness that it needs. This is what we typically hear described as *agility*: the ability to react intelligently to stimuli and also continually reshape the reaction as circumstances change.

Data is powerful, if you can see it and know what to do with it. To paraphrase Levitt and Dubner and their great book, *Freakonomics*, an EDA provides potential adaptation data that exists in streams that we can't possibly see on our own. Levitt and Dubner characterize the Internet as a "gigantic horseshoe magnet waved over an endless sea of haystacks, plucking the needle out of each one." Similarly, an EDA—the nervous system—gives us a way to acquire data and to make the data we have meaningful. For example, if we knew that every day we experience what it feels like to have a lion stepping on our toe, followed by no negative reaction, we might learn to ignore it as unimportant—or begin to assume that it's not a lion. That's fine, until a lion does step on our toe…

Building an EDA to instill good functioning to the enterprise nervous system involves getting the various sensors, message pathways, and reacting logic processors to work together. In broad terms, this is known as interoperation, and it is the heart of the new EDA discussion going on today.

The "New" Era of Interoperability Dawns

Reading about EDA as a "new" idea might give you a sense of déjà vu. As we saw with the familiar credit card fraud example, EDA is not a new concept. However, the current crop of EDAs uses proprietary standards for communication, and although they work well, they are, in effect, tightly coupled EDAs that can only share information among systems that use a compatible standard. For instance, it is possible to set up a fairly effective EDA if all systems are built on the same platform. Vendors have long provided high-performance pub/sub engines for compatible systems. The only problem is, as we know, not everyone is on the same platform, despite the dramatic sales efforts of some of Silicon Valley's best and brightest minds. The good news is that many platform vendors have released new service-based EDA products, which do not rely on tight coupling.

The quest for a well-functioning enterprise nervous system has been the catalyst for the development of EDA for many years. Why, then, is EDA receiving such renewed and intense interest today? The reason has to do with the explosion in interoperability and the standardization of data across multiple enterprises, which changes the game of EDA.

Ultimately, the existence of an EDA is dependent on interoperability among systems. You can't have awareness and reaction to events if the systems cannot communicate with one another. Existing EDA setups are invariably tightly constrained and narrow in their functionality because it has been so difficult, or costly, to achieve the level of interoperation of EDA components needed for any kind of dynamic or complex EDA functionality. That is now changing. Today, with the advent of open standards and the breakthroughs in system interoperability from service-oriented architectures (SOAs), it is now possible to establish EDAs that are far more intelligent, dynamic, and far-reaching than ever before.

To put the interoperability evolution in context, we will share a lunch conversation we had recently with a man who had been responsible for designing and implementing the basic underpinnings of the worldwide airline reservation and automated teller machine infrastructures. In the last few years, he has been involved in other pursuits, so he was eager to learn about SOA and EDA, the new tech trends that he had been hearing so much about in the industry media.

When we explained how the related concepts of SOA and EDA allowed, for the first time ever, truly open interoperation among heterogeneous software applications, regardless of operating system, network

protocol, or programming language, he gave us a perplexed look. "That's new?" he asked with a slight smirk. "That idea has been around since 1961."

And, of course, he was right. The idea of open interoperability has been in the air for decades. Just like the automobiles have been around since the late 1800s. Even today, we still use a combustion engine to drive the wheels, so the essence hasn't changed much. However, never before have cars been so reactive to our needs and their surroundings.

The same evolution is true for the software and the circumstances that we find ourselves in now, in 2009. Over the last eight years, from 2001 to 2009, we have seen an unprecedented shift in the IT industry toward the use of open standards for the purpose of integrating diverse software applications. Also, more and more companies are exposing their data in a standardized fashion further expanding the circle of opportunity.

This all started back in 2001…. An unusually broad group of major IT companies, including IBM, Oracle, Microsoft, BEA, and others, agreed to conform to a specific set of Extensible Markup Language (XML) standards for interoperation between software applications. These standards, known collectively as the Web Services protocol, provide a technological basis for any application in the world to exchange data or procedure calls with any other application, regardless of location, network, operating system, or programming language.

Specifically, the major standards that were ratified included Simple Object Access Protocol (SOAP), which is the message formatting standard, Web Services Description Language (WSDL), which sets out a standard document format with which to describe a Web service, and Universal Description, Discovery, and Integration (UDDI), a Web services registry application programming interface (API), that are available for use in a particular domain.

Thus, Web services are software-based interfaces that are universally understandable, self-describing, and universally discoverable. As our colleague Jnan Dash, the legendary lead engineer of the Oracle Database, puts it, the combination of the Internet and Web services makes possible a kind of "universal dial tone" for all applications. (The Internet is the dial tone and Web services give you the ability to "dial.") With Web services, it is entirely possible for an application written in C to interoperate with a J2EE application without the need for any proprietary middleware. As a whole, the large-scale development and integration of Web services is a key step toward developing a service-oriented

architecture (SOA). SOA represents a model in which functionality is decomposed into small, distinct units (services—for example, Web services), which can be distributed over a network and can be combined together and reused to create business applications. These services communicate with each other by passing data from one service to another or by coordinating an activity between two or more services.[2] The industry vision that is fueling the SOA trend is that one day virtually any application needed in your enterprise (whether it is inside your firewall or not) will be available as a Web service and will be freely interoperable with other applications, enabling the decomposition of application functionality into small units that can interoperate or be orchestrated in composite applications. This vision is idealized, and it is likely that a full-blown SOA of this type will never actually come into existence. However, many are approaching the paradigm in steps.

Many enterprises have begun to introduce service-orientation in their architectures, selectively exposing capabilities through Web services in configurations that suit specific business needs and selectively service enabling core legacy systems. This is a remarkable achievement for an industry that was very much in the doghouse after the Y2K panic and dot.com fiascos of 2001. The most striking thing about SOA, beyond the fact that Web services standards were adopted simultaneously by many large IT vendors, is the fact that it actually works. SOA is very much the technological trend of the moment, and it is everywhere. You see SOA as a prominent feature set in products from Microsoft, Oracle, IBM, SAP, and so on. Virtually every major technology company has announced an SOA strategy or even shifted their entire market focus to being service-oriented. A sure sign that SOA had reached prime time was when Accenture announced that it was going to spend $450 million on an SOA consulting initiative for its global clients.

SOA removes much, if not all, of the proprietary middleware and network compatibility blockages that inhibit rapid changes in application integration. As a result, they can loosen the coupling between applications. Given how important agility is, tight coupling is rightly held out as the enemy of agility. Loose coupling is the enabler of agility and SOA delivers loose coupling. Changes become simpler, faster, and cheaper. As integration agility becomes reality, so does EDA and its increased awareness. Therefore, SOA delivers the necessary agility required for an EDA. However, achieving this goal of EDA through loose coupling without destroying a range of security, governance, and performance

standards requires a great deal of planning and work. And, as we start to see, the path from where we are now, to SOA and then EDA, is not always clear.

The ETA for Your EDA

This is not a cookbook, but it can put you on track for finding the right use for EDA in your organization and getting it started. At the very least, our intent is to familiarize you with this exciting new technological paradigm—and you will need this familiarity if you are a professional working in technology today. EDA and SOA are appearing in a myriad of commercial IT offerings and technological media articles. You need to know about EDA.

How you approach EDA is up to you, and if your career has been like ours, you might agree that rushing is seldom a good idea, especially when a new technology is involved. Our goal is to inform and stimulate your thinking on the subject. Whatever the ETA is for your EDA, only you will know the right way to proceed. Our wish is to give you the knowledge and insight you need to make it a success.

Endnotes

1. Michelson, Brenda. "Event Driven Architecture Overview." Paper published by Patricia Seybold Group (2/2/2006).
2. Wikipedia, http://en.wikipedia.org/wiki/Service-oriented_architecture.

CHAPTER 3

Characteristics of EDA

Firing Up the Corporate Neurons

Getting to a complete understanding of event-driven architecture (EDA) takes us on a step-by-step process of learning. First, we discussed the enterprise nervous system and the way EDAs are formed by connecting event listeners with event consumers and event processors, and so on. Then, to explain how these EDA components will likely be realized in today's enterprise architecture, we learned about Web services and service-oriented architecture (SOA). To get the full picture, though, we now need to get into depth on the characteristics and qualities of EDA components.

If the EDA components are like the neurons in the enterprise nervous system, then we need to understand how their "synapses" and neural message pathways work if we want to form a complete picture of EDA. We need to know how they actually can or should work together to realize the desired functionality of an EDA. In this context, we go more deeply into the concept of *loose coupling* and also explore in depth the ways that an EDA needs to handle messaging between its components. With the key concepts defined, we then lay out a thorough definition of EDA, using an idealized EDA as an example.

Revisiting the Enterprise Nervous System

Returning to our cat scenario, if your cat steps on your toe, how do you know it? How do you know it's a cat, and not a lion? You might want to pet the cat, but shoot the lion. And, perhaps most important, how can you be sure that your body and mind learn how to distinguish between

the cat and the lion in the first place? How do you keep learning to process sensory experiences? The world is constantly changing, so our nervous systems, and our EDAs, must be flexible, adaptable, and fast learners. We want our EDAs to be as sensitive, responsive, and teachable as our own nervous systems. To get there, we need to endow our EDA components with nervous system–like capabilities.

When the cat's paw presses against your toe, the nerve cells in your toe fire off a signal to your brain saying, "Hey, something stepped on my toe." In this way, the neurons in your toe are like event producers. The neural pathways that the messages follow as they travel up your spine to the brain are like the messaging backbone of the EDA. Your brain is at once an event listener and an event processor. If you pet the cat, your hand and the nerves that tell your hand to move are event reactors. Figure 3.1 compares the EDA with your nervous system.

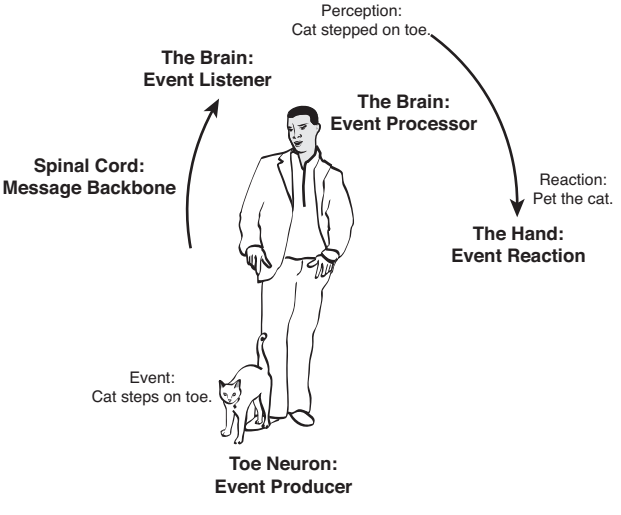

Figure 3.1 The human nervous system compared with an EDA.

The nervous system analogy is helpful for getting the idea of EDA on a number of levels. In addition to being a useful model of the EDA components in terms that we can understand (and perhaps, more important, that you can use to explain to other less-sophisticated people), we can learn a lot about how an EDA works by understanding how the nerves and brain communicate and share information. As a first step in mapping from nervous system to EDA in terms of its characteristics, we

look at event-driven programming, a technology that is comparable to an EDA and quite familiar, as well as informative.

Event-Driven Programming: EDA's Kissing Cousin

We all use a close cousin of EDA on a daily basis, one whose simplicity can help us gain a better understanding of EDA, perhaps without even realizing it. It's called event-driven programming (EDP) and it's common in most runtime platforms. It's also found in CPU architectures, operating systems, GUI interfaces, and network monitoring. EDP consists of event dispatchers and event handlers (sometimes called event listeners). Event handlers are snippets of code that are only interested in receiving particular events in the system. The event handler subscribes to a particular event by registering itself with the dispatcher. The event dispatcher keeps track of all registered listeners then, when the event occurs, notifies each listener through a system call passing the event data.

For example, you might have a piece of code that executes if the user moves the mouse. Let's call this a mouse event listener. As shown in Figure 3.2, the mouse event listener registers itself with the dispatcher—in this case, the operating system. The operating system records a callback reference to the mouse event listener. Every time the user moves the mouse, the dispatcher invokes each listener passing the mouse movement event. The mouse movement event signals a change in the mouse or cursor position, hence a change in the system's state. Other examples of event-driven programming can be found in computer hardware interrupts, software operating system interrupts, and other user interface events, such as mouse movements, key clicks, text entry, and so on.

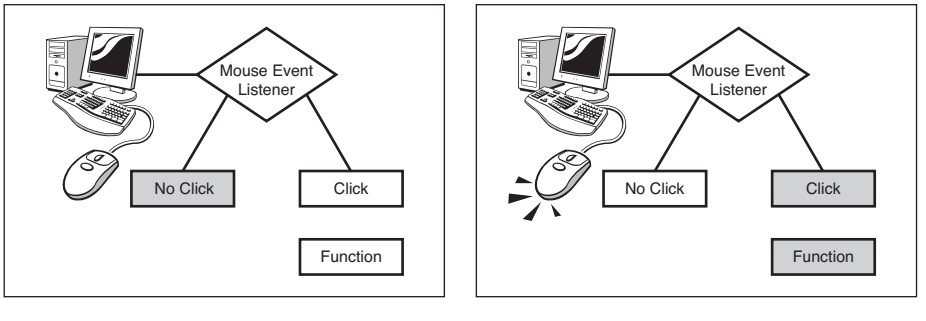

Figure 3.2 The PC's instruction to listen for mouse clicks is an example of event-driven programming (EDP), a close cousin of EDA. When the mouse is clicked, the mouse click event listener in the PC's operating system is triggered, which, in turn, activates whatever function is meant to be invoked by the mouse click. When the mouse is not clicked, the event listener waits.

Wikipedia describes event-driven programming as, "Unlike traditional programs, which follow their own control flow pattern, only sometimes changing course at branch points, the control flow of event-driven programs [is] largely driven by external events."[1] The definition points out that there is no central controller of the flow of data, which is counterintuitive to the way most of us were taught to program.

The reason we bring this up is to emphasize a key distinction between EDP and conventional software: a lack of a central controller. This distinction is critical to understanding how EDA works. When you first enter the programming world, you're taught how to write a "Hello World" program. You might learn that a program has a main method body from which flow control is transferred to other methods. The main method is treated like a controller (see Figure 3.3).

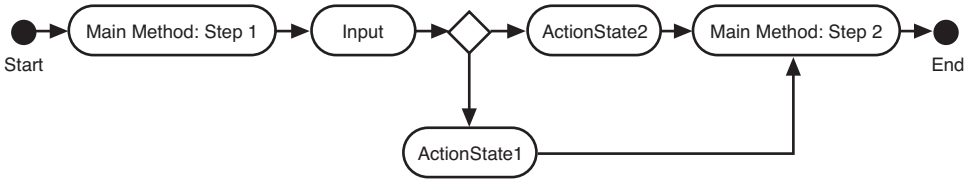

Figure 3.3 In a conventional programming design, a controller method controls the flow of data and process steps.

In contrast, in event-driven programming, there are no central controllers dictating the sequence flow. As shown in Figure 3.4, each component listening for events acts independently from the others and often has no idea of its coexistence. When an event occurs, the event data is relayed to each event listener. The event listener is then free to react to that information however it chooses, perhaps activating a process specifically intended for that particular event trigger. The event information is relayed asynchronously to the event listeners so multiple listeners react to the event data at the same time, increasing performance but also creating an unpredictable order of execution.

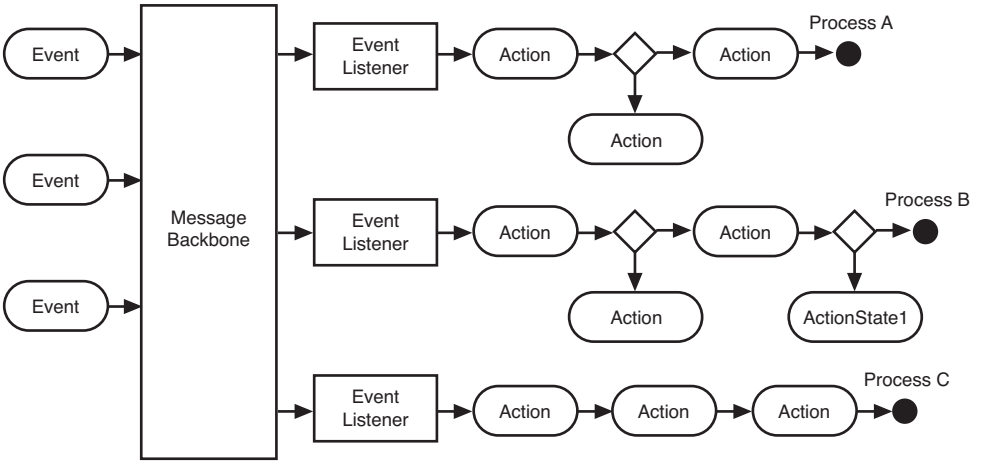

Figure 3.4 In event-driven programming (EDP), event listeners receive state change data (events) and pass them along to event dispatchers, which then activate processes that depend on the nature of the triggering events.

As shown in Figure 3.4, the listeners execute concurrently. This is quite different from the typical program that controls the flow of data. In a typical program, the controller method calls out to each subcomponent, passes relevant data, waits for control to return, then continues to the next one—a very predictable behavior. Of course, the controller method could take an asynchronous approach, but the point is that one has a predefined flow of data whereas the other does not.

When waiting for events, event listeners are typically in a quiescent state, though occasionally you'll see a simulated event-driven model where event listeners cyclically poll for information. They sleep for a predefined period then awaken to poll the system for new events. The sleep time is usually so small that the process is near real time.

Similar to EDP-based systems, EDA relies on dynamic binding of components through message-driven communication. This provides the loose coupling and asynchrony foundation for EDA. EDA components connect to a common transport medium and subscribe to interested event types. Most EDA components also publish events—meaning they are typically publishers *and* subscribers, depending on context. The biggest difference between EDA and EDP is that EDP event listeners are colocated and interested in low system-level events like mouse clicks, whereas EDA event consumers are likely to be distributed and interested in high-level business actions such as "purchase order fulfilled."

More on Loose Coupling

Let's go deeper on loose coupling, a core enabling characteristic of EDA. You can't have EDA without loose coupling. So, as far as we EDA believers are concerned, the looser the better. However, getting to an effective and workable definition of loose coupling can prove challenging. If you ask nine developers to define loose coupling, you'll likely get nine different answers. The term is loosely used, loosely defined, and loosely understood. The reason is that the meaning of *loose coupling* is context sensitive. For EDA purposes, loose coupling is the measurement of two fundamentals:

- Preconception
- Maintainability (Changeability)

Preconception: The amount of knowledge, prejudice, or fixed idea that a piece of software has about another piece of software

Preconception is a quality of software that reflects the amount of knowledge, prejudice, or fixed idea that one piece of code has about another piece of code. The more preconception that an application (or a piece of an application) has in relation to another application with which it must interoperate, the tighter the coupling between the two. The less preconception, the looser the coupling. We've all seen tight coupling that stems from high levels of preconceptions. Think of systems where every configuration attribute and every piece of mutable text is hard-coded in the system. It can take days just to correct a simple spelling mistake. During design, these systems all made a single, yet enormous, configuration preconception—they assumed that the configuration would be set at compile time and never need to be changed. You will never get to the flexibility of configuration that you need to build an EDA with this kind of tight coupling.

Ultimately, to move toward EDA and SOA, you should strive for software that makes as few presumptions as possible. To use a common, real-world example of tight coupling, consider a point-of-sale (POS) program calling a credit card debit (CCD) program and passing it a credit card (CC) number. As shown in Figure 3.5, the POS program has a preconceived notion that it will always be calling the CCD program and always be passing it a CC number, hence the two systems are now tightly coupled.

Maintainability: The level of rework required by all participants when one integrated component changes

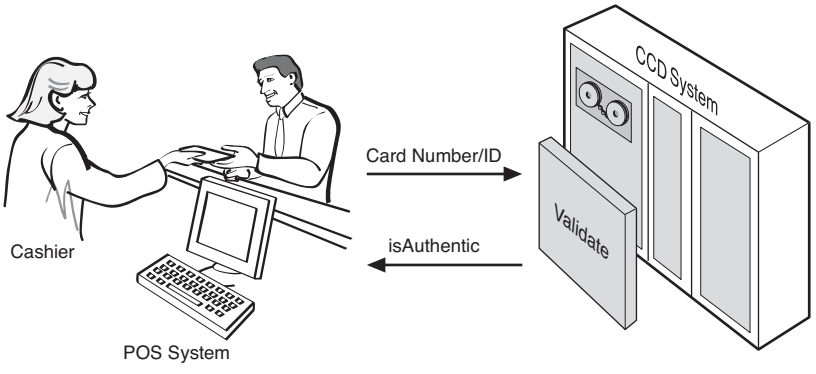

Figure 3.5 In this classic example of tight coupling, a POS system sends a credit card number to a CCD program and requests a validation, which is indicated by a returned value of `isAuthentic`. The two systems are so tightly bound together they can almost be viewed as one single system.

Maintainability, the other EDA-enabling component of loose coupling, refers to the level of rework required by all participants when one integrated component changes. When a piece of software changes, how much change does that introduce to other dependent software pieces? Best practices dictate that we should strive for software that embraces and facilitates change, not software that resists it. As a rule, the looser the coupling between components or systems, the easier it is to make software changes without impacting related components or systems.

Consider the hard-coded POS system described previously. A simple configuration change requires a source code change, compilation, regression testing, scheduled system downtime, downtime notifications, promotion to production, and the like. A system that resists change is considered a tightly coupled system.

What Your CFO Is Thinking

Imagine that you are the owner of this tightly coupled POS system, and your CFO tells you that, as of some very rapidly approaching date, she expects the POS systems to accept coupons as a form of payment in addition to credit cards. Unlike the credit cards, which have a 16-digit identifying number and a matching expiration date, the coupons have a 10-character identifier composed of letters and numbers. When you tell your CFO that it might take you three months to make this change, she is not going to be too interested in the issues of maintainability and preconception involved in the POS software, but you know that these two tight coupling demons are to blame. The coupons might actually provide you with a good pretext to start discussing an EDA/SOA approach to POS. You can tell the CFO that you can make future coupon transitions faster if you loosen up the coupling in the POS systems.

Now let's suppose we begin to alleviate our headaches by removing some of the system's preconceived ideas. As a start, let's assume we make the following two changes:

- First, we remove the hard-coded instructions from our system code, and instead let behavior be driven by accessing values stored in a configuration file (presumably read into memory at instantiation).
- Second, we enable our system to be dynamically reconfigured (meaning our system would have a mechanism for reloading new versions of the configuration file while still active).

In this case, making a simple change to our configuration file, such as indicating that an entry in coupon format is a valid form of payment or even correcting a spelling mistake, only requires a regression test and a signal sent to the production system to reload its configuration. The system is *maintainable*—we updated the system while it stayed in production, and we did so without compiling a lick of code.

We have also successfully decreased the coupling between our system and its configuration. The system is now loosely coupled with respect to this context but it might still be tightly coupled in other areas. We have only increased its loosely coupled index. We have increased its changeability and decreased its preconception with respect to configuration, but how does it interact with other modules or components? It might be tightly coupled with other software.

This is where the meaning of loose coupling is context sensitive. We can say the system is loosely coupled if that statement is made within the context of the configuration file. We can also say it is not loosely coupled if the statement is made referring to its integration techniques.

This example oversimplifies the situation because hard-coded systems are often very difficult to modify into configuration-driven systems, and even harder to modify to dynamically configuration-driven systems, but the points are valid. We did decrease the tight coupling and ease our headache. Moreover, we can see that significant rework time would have been saved had the system designers taken this approach from the beginning.

To illustrate our point, we have just used an example where we increased the degree of loose coupling of the system by loosely coupling configuration attributes. However, the term is typically used to reference integration constraints. Two or more systems are tightly coupled when their integration is difficult to change because of each system's preconceptions.

Our previous point-of-sale (POS) scenario is an example of two tightly coupled systems. Changes in either system are very likely to necessitate changes in the other. At the extreme (though not uncommon) end of this spectrum, the overall design might be so tightly integrated that the two systems might be considered one atomic unit.

The POS system has preconceived notions about how to interact with the CCD system. For example, the POS system calls a specific method in the CCD system, named `validate`, passing it the CC number. Now suppose the CCD system changes the method name to `isAuthentic`. This might happen if a third party purchased the CCD system, for example.

What we want to do is isolate those changes so that we do not have to change our POS system with every vendor's whim. To loosen up the architecture, let's exercise a design pattern called the **adapter** pattern. We will add an intermediate (adapter) component between the POS and CCD systems. The sole purpose of this component is to isolate the preconceived knowledge of the CCD system. This allows the vendor to make changes without adversely affecting the POS system.

Now vendor changes in the CCD system are isolated and can be bridged using the intermediate component. As the diagram in Figure 3.6 illustrates, the vendor can change the method name and only the adapter component needs to change.

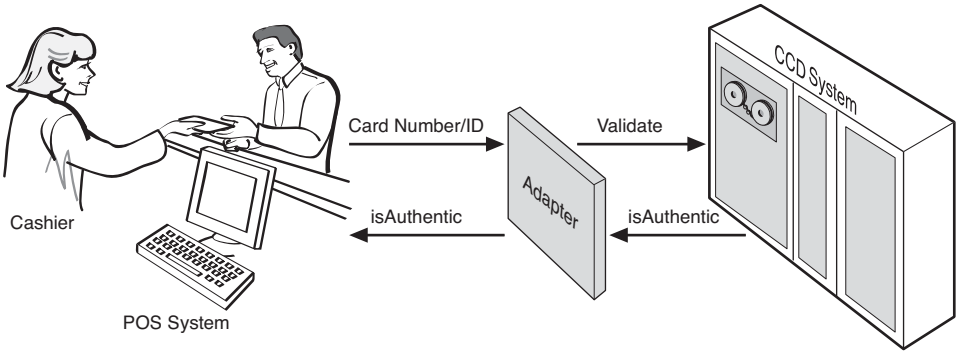

Figure 3.6 The insertion of an adapter between the POS and CCD systems loosens the coupling. Changes to the CCD system are isolated and can be bridged using the adapter.

This reduces the POS system's preconception about the CCD giving the systems greater changeability. In essence, we now have greater business flexibility because we now have the freedom to switch vendors if we choose. We can swap out the Credit Card Debit (CCD) product for another just by changing the adapter component.

The true benefits of the design shown in Figure 3.6 are radically evident when we talk about multicomponent integration, which is shown in Figure 3.7. Here, the benefits are multiplied by each participating component. This is also where the return on investment shows through reuse. Understand that the up-front time spent on building the adapter is now saving more money with each use. The more you use it, the more you'll save.

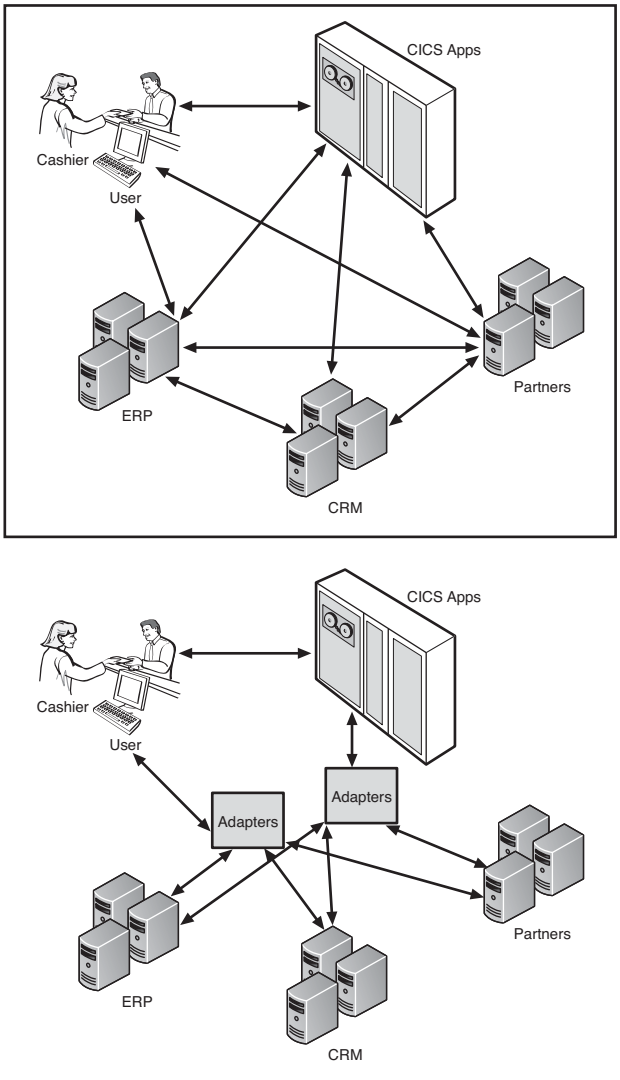

Figure 3.7 Use of adapters in multicomponent integration.

The argument can be made that we have now only shifted the tight coupling to our adapter, which is true, though we have added a layer of abstraction that does, in fact, increase maintainability of the system. We'll demonstrate how to fully decouple these systems when we talk about event-driven architecture later in this chapter.

There will always be a degree of coupling. Even fully decoupled components have some degree of coupling. The desire is to remove as much as possible but it is naïve to think the systems will ever be truly decoupled. For example, service components need data to do their job, and as such will always be coupled to the required input data. Even a component that returns a time stamp is tightly coupled with the system call used to retrieve the current time. As we strive for loose coupling, we should remember that the best we can achieve is a high "degree of looseness."

More about Messages

Coupling, loose or tight, is all about messages. For all practical purposes, it is only possible to have loose coupling and EDA, with a messaging design that decouples the message sending and receiving parties and allows for redirection if needed. To see why this is the case, let's look at two core aspects of messaging: harmonization and delivery. Harmonization is how the components interact to ensure message delivery. Delivery is the messaging method used to transfer data.

Message harmonization is how the components interact to ensure message delivery.

Harmonization can be synchronous or asynchronous. Synchronous messaging is like a procedure call shown in Figure 3.8. The producer communicates with the consumer and waits for a response before continuing. The consumer has to be present for the communication to complete and all processing waits until the transfer of data concludes. For example, most POS systems and ATMs sit in a waiting state until transaction approval is granted. Then, they spring back into life and complete the process that stalled as the procedure call was completed. Comparable examples of synchronous messaging in real life include instant messaging, phone conversations, and live business meetings.

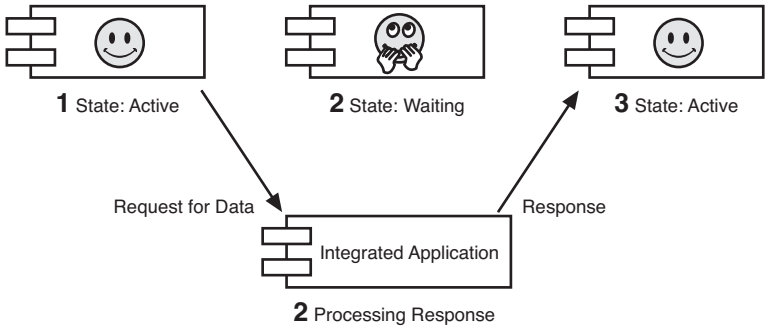

Figure 3.8 Example of synchronous messaging, a process where the requesting entity waits for a response until resuming action.

In contrast, asynchronous messaging does not block processing or wait for a response. As Figure 3.9 illustrates, the message consumer in an asynchronous messaging setup need not be present at the time of transmit. This is the most common form of communication in distributed systems because of the inherent unreliability of the network. In asynchronous messaging, messages are sent to a mediator that stores the message for retrieval by the consumer. This allows for message delivery whether the consumer is reachable or not. The producer can continue processing and the consumer can connect at will and retrieve the awaiting messages. Examples include e-mail (the consumer does not need to be present to complete delivery), placing a telephone call and leaving a voice mail message (versus a world without voice mail), and discussion forums.

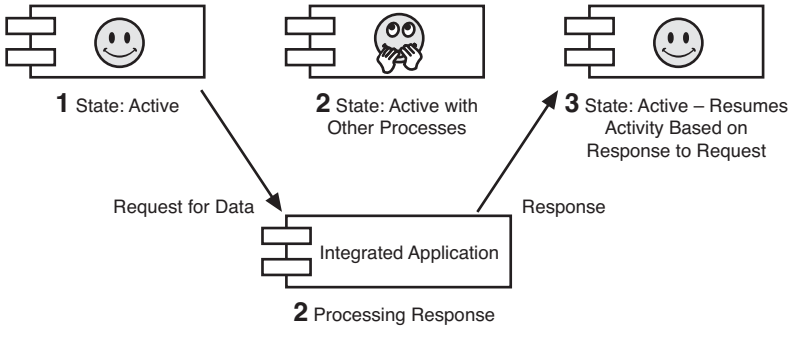

Figure 3.9 Example of simple, point-to-point asynchronous messaging.

There are multiple ways to execute message delivery whether synchronously or asynchronously. Synchronous messaging includes request/reply applications like remote procedure calls and conversational messaging like many of the older modem protocols. Our focus here is on asynchronous messaging. Asynchronous messaging comes in two flavors: point-to-point or publish/subscribe.

Message delivery is the messaging method used to transfer data.

Point-to-point messaging, shown in Figure 3.9, is used when many-to-one messaging is required (meaning one or more producers need to relay messages to one consumer). This is orchestrated using a queue. Messages from producers are stored in a queue. There can be multiple consumers connected to the queue but only one consumer processes each message. After the message is processed, it is removed from the queue. If there are multiple consumers, they're typically duplicates of the same component and they process messages identically. This multiplicity is to facilitate load balancing more than multidimensional processing.

Publish/subscribe messaging, shown in Figure 3.10, is used when many applications need to receive the same message. This wide dissemination of event data makes it ideal for event-driven architectures. Messages from producers are stored in a repository called a topic. Table 3.1 summarizes the differences between the two modes of message flow. Unlike point-to-point messaging, pub/sub messages remain in the topic after processing until expiration or purging. Consumers subscribe to the topic and specify their interest in currently stored messages. Interested consumers are sent the current topic contents followed by any new messages. For others, communication begins with the arrival of a new message.

Topics provide the advantage of exposing business events that can be leveraged in an EDA. One consideration is the transaction complete indeterminism, and we will soon explore ways to handle this.

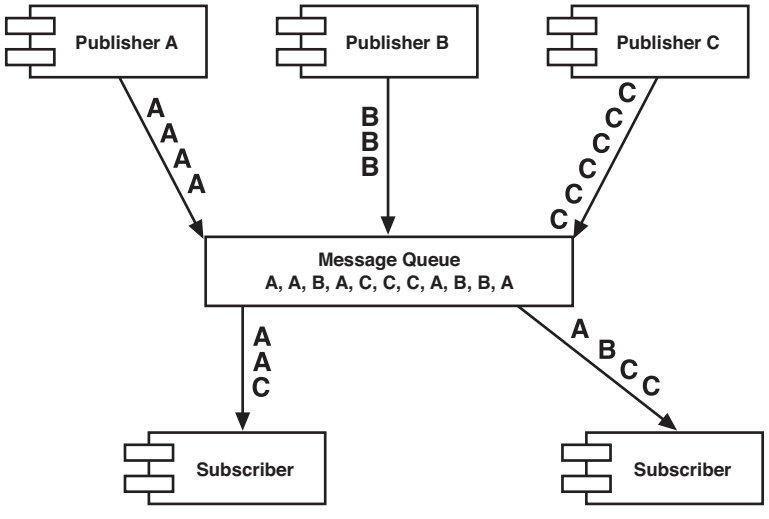

Figure 3.10 Example of publish/subscribe (pub/sub) asynchronous messaging using a message queue.

Asynchronous messaging requires a message mediator, or adapter. This can be achieved using a database, native language constructs like Java Channels, or the most common provider of this functionality, message-oriented-middleware (MOM). MOM software is a class of applications specifically for managing the reliable transport of messages. This includes applications like IBM's WebSphere MQ (formally MQSeries), Microsoft Message Queuing (MSMQ), BEA's Tuxedo, Tibco's Rendezvous, others based on Sun's Java Messaging Specification (JMS), and a multitude of others.

JMS is the most prominent vendor-agnostic standard for message-oriented-middleware. Before its creation, messaging-based architectures were locked in to a particular vendor. Now, most MOM applications support the standard, making it the primary choice for implementation teams concerned with vendor-agnostic portability.

Table 3.1 Point-to-Point Versus Publish/Subscribe

Point-to-Point Queues	Publish/Subscribe Topics
Single consumer	Multiple consumers
Preconceived consumer	Anonymous consumers
Medium decoupling	High decoupling
Messages are consumed	Messages remain until purged or expiration

The Ideal EDA

Having taken our deep dive into the key characteristics of EDA, we can now examine a workable, if idealistic definition of EDA. With the usual caveat that no architecture will, in all likelihood, ever embody EDA in 100% of its functionality, we can define EDA as an enterprise architecture that works in the following ways:

EDA: What It Is

- An EDA is loosely coupled or entirely decoupled.
- An EDA uses asynchronous messaging, typically pub/sub.
- An EDA is granular at the event level.
- EDAs have event listeners, event producers, event processors, and event reactors—ideally based on Simple Object Access Protocol (SOAP) Web services and compatible application components.
- An EDA uses a commonly accessible messaging backbone, such as an enterprise service bus (ESB) as well as adapters or intermediaries to transport messages.
- An EDA does not rely on a central controller.

EDA: What It Does and What It Enables

- An EDA enables agility in operational change management.
- An EDA enables correlation of data for analytics and business process modeling, management, and governance.

- An EDA enables agility in realizing business analytics and dynamically changing analytic models.
- An EDA enables dynamic determinism—EDA enables the enterprise to react to events in accordance with a dynamically changing set of business rules, for example, learning how to avoid shooting the cat and petting the lion (in contrast to controller-based architectures that can be too rigid to be dynamic, for example, shooting the cat, not being aware of the lion).
- An EDA brings greater consciousness of events to the enterprise nervous system.

Though we delve more deeply into the ways that SOAP Web services enable EDA later in the book, we want to go through a basic explanation at this point because our described use of Web services as event producers might appear confusing to some readers. Much has been written about Web services in recent years, and, indeed, many of you likely already work with them. It might seem incorrect to characterize a Web service as a "producer" of SOAP Extensible Markup Language (XML) event state messages when Web services, to be accurate, actually respond to invocation, perhaps sending off SOAP XML if instructed to do so. This is, of course, correct. A SOAP Web service does not transmit a SOAP message without being triggered to do so. Thus, when we talk about Web services functioning as event producers, we are describing Web services that are specifically programmed to send event data to the message backbone. These event Web services could be triggered by activities occurring inside an application or by other Web services. The reason we suggest that event producers should be configured in this way—as Web services that transmit event state data upon invocation—is that there is a high level of utility in transmitting the event data in the portable, universally readable SOAP XML format.

Figure 3.11 revisits our phone company example and shows a high-level model of how its systems would function and interoperate in an ideal EDA. Let's make a few basic observations about how the company's EDA works. With an EDA, in contrast with the traditional enterprise application integration (EAI) approach, the company's three system groups all send event data through adapters and message listeners to a service bus, or equivalent EDA hub that manages a number of pub/sub message queues for all systems that need that event data to carry out their tasks.

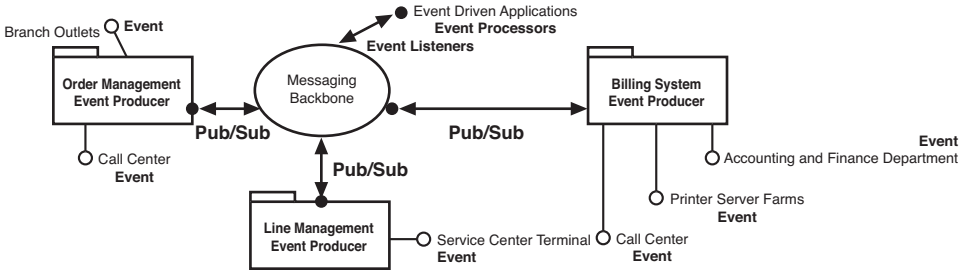

Figure 3.11 A high-level overview of an event-driven architecture at a phone company. Each system group is loosely coupled with one another using standards-based pub/sub asynchronous messaging.

As shown in Figure 3.12, using a dynamic determinism model, the order management system can now listen for overages in minutes and unpaid bills that occur in billing and line management system events and respond to them according to the business rules. Thus, if "John Q" exceeds his allowance of wireless minutes and fails to pay for the overage, the business rules contained in the order management system will deny him the right to add new services to his account.

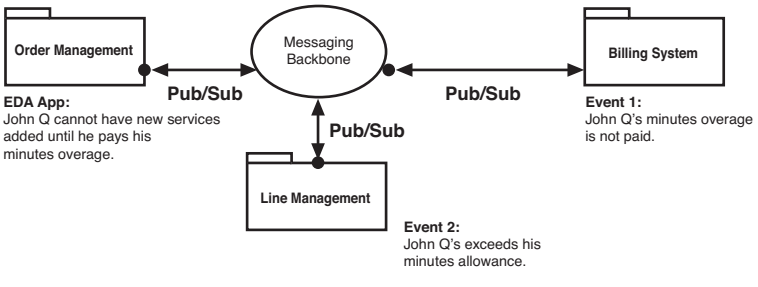

Figure 3.12 In the phone company EDA example, separate events in two systems—an overage in wireless minutes and an unpaid balance in the billing system—are correlated by an application that then denies the order management system the ability to grant the customer a new service request.

The order management system does not have to have the kind of preconception about the line management system that it needed to have to provide this function under the EAI model. The two systems are decoupled but still interoperating through the EDA. The billing system is the event producer and the order management system is the event consumer.

Figure 3.13 shows how event listeners detect the two separate events—the unpaid overage charge and the overage in minutes itself. The EDA-based application that authorizes or declines the new service request subscribes to the event publishing done by the line management and billing systems. The combination of events—unpaid balance and overage of minutes—combines to change the state of John Q's account. The change in state is itself an event. John Q's status goes from "eligible" to "ineligible" for new services. If John Q requests new services, the order management system looks to the EDA application to determine if John Q's status is eligible.

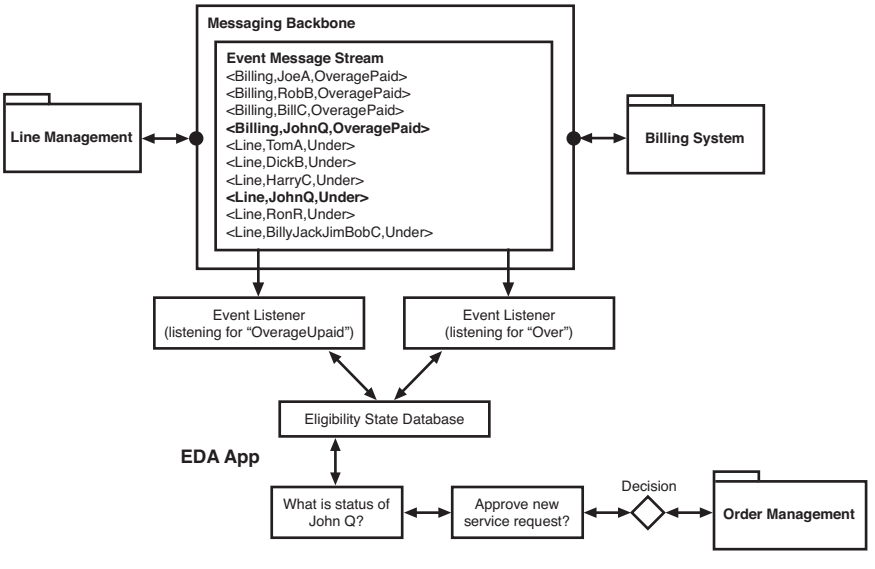

Figure 3.13 The EDA-based application subscribes to event data that is published and consumed by event listeners on separate systems. This gives the EDA-based application the ability to have awareness of changes in state related to John Q without tightly coupling any of the applications involved in the query.

EDA opens new worlds of possibility for IT's ability to serve its business purpose. Think of all the business events a system could leverage if events were exposed—examples include events such as order processing complete, inventory low, new critical order placed, payment received, connection down, and so on. Today, it's a struggle to expose the needed events because they're hidden away within legacy systems. It's common to resort to database triggers or polling to expose these critical actions,

but imagine the supportable agility if the systems exposed those actions natively.

Exposing system actions is the root of most integration complexities. "Upon completion of processing at System A, send result to system B," and so on. Most legacy systems were not designed with unanticipated use in mind. They assumed they would be the only system needing the information and thus didn't expose key event data for easy access. If you're lucky, the system will provide an application programming interface (API) to retrieve data, but rarely will it facilitate publishing an event or provide any event retrieval mechanism. Because events are typically not exposed, the first thing you have to do is create an algorithm to determine an event occurred. Often, legacy system events have to be interpreted by correlating multiple database fields (e.g., "If both of these two fields change state, then the order has been shipped…"). Imagine how much easier integration would be if such event actions were natively exposed.

Event-driven architectures are driven by system extensibility (not controllability) and are powered by business events. As shown in Figure 3.13, event handlers listen to low-level system events while EDA agents respond to coarser-grained business events. Some agents might only respond to aggregate business events, creating an even coarser system response.

EDAs are based on dynamic determinism. Dynamic determinism relates to unanticipated use of applications and information assets. Events might trigger other services that might be unknown to the event publisher. Any component can subscribe to receive a particular event unbeknown to the producer. Because of this dynamic processing, the state of the transaction is managed by the events themselves, not by a management mechanism.

EDA embraces these concepts, which facilitate flexibility and extensibility, ultimately increasing a system's ability to evolve. This is accomplished through calculated use of three concepts—loose coupling, asynchrony, and stateless (modeless) service providers—though it doesn't come free. EDA brings inherently decentralized control and a degree of indeterminism to the system.

One of the main benefits of EDA is that it facilitates unanticipated use through its message-driven communication. It releases information previously trapped within monolithic systems. When designing EDA components, you should design for unanticipated use by producing

events that can provide future value whether a consumer is waiting or not. Your EDA components should be business-event-intuitive, publishing actions that are valued at a business level.

Imagine an EDA billing component. After it has finished billing a customer, it should announce the fact even if there is no current need. What if all financial actions were being sent via events? Recognizing that there was no immediate need for these events when the systems were originally built, look at how beneficial it would be today. Imagine how easy that would have made your company's Sarbanes-Oxley compliance efforts. Of course, it takes a degree of common sense in determining what might be of value in the future, but it's safe to say that most concrete business state changes will be valued. The caution to note here, though, is that it is possible to create an event publishing overload that overwhelms system and network capacity.

EDA components should also be as stateless as possible. The system state should be carried *in the event*, not stored within a component variable. In some situations, persistence is unavoidable, especially if the component needs to aggregate, resequence, or monitor specific events. However EDA components should do their job and pass on the data then return to process or wait for the next event. This gives the system ultrahigh reuse potential and flexibility. The flexibility of an EDA is leading to emerging concepts that leverage events at a business process level.

Consciousness

EDA brings consciousness to the enterprise nervous system. Without event-driven architecture (EDA), enterprises operate as if they're on life support. They're comatose (brain dead), meaning they are unaware of their surroundings. They cannot independently act on conditions without brokered instruction or the aid of human approval. Service-oriented architectures (SOAs) define the enterprise nervous system, while EDA brings awareness. With the right mix of smart processing and rules, EDA enables the enterprise nervous system to consciously react to internal and external conditions that affect the business within a real-time context.

Consciously reacting means the architecture acts on events independently without being managed by a central controller. Underlying components react to business events in a dynamic decoupled fashion. This is in contrast to the central controller commonly seen in SOAs.

Imagine the analogy of our consciousness with a cluster of functional components. Sections of consciousness process certain information, just

like each component has an area of expertise. Components wait for pertinent information, process, and fire an output event. The output might be destined to another component or to an external client. Our consciousness works in the same manner, processing information and sending output to either other synaptic nodes or externally, perhaps through vocal communication. In both of these cases, the messages were not sent to a central controller to decide where to route or what to do. The behavior is inherent in the design.

This is in direct contradiction to the way we teach and learn to program. Schools and universities teach us to start every project with a central controller. In Java, this would be the *main* method, where the sequence of control and the flow of information are controlled. This type of system is tightly coupled with the controller and is difficult to make distributed. Today's architectures need to be looser coupled and more agile than we've been taught.

Today's systems need true dynamic processing. Systems are classified as dynamic or static, but, in reality, most systems are static; they have a finite number of possible flows. If a system has a central controller, it's definitely static even if control branches are based on runtime information. This makes testing easier because of the degree of predetermination but does not provide the agility of a dynamic system.

A central controller with a limited number of possibilities decreases agility. When the system needs to change outside of those possibilities, new rules and branches are added, increasing the tight coupling and complicating the architecture. Over time, the branching rules become so complex that it's nearly impossible to manage and the system turns legacy.

EDA is about removing the rigidity created by central control and injecting real-time context into the business process.

We need to be clear about one thing here: When we talk about removing central control, we are not suggesting that you can be effective in an EDA by removing all control from the application. An uncontrolled application would quickly degenerate into chaos and lock itself up in inaction, or in inappropriate action. Real-world autonomic systems see this: Three moisture-ridden sensors in a B-2 bomber sent bad data to the aircraft's computer, causing it to fly itself into the ground. Another example is the human body's response to significant blood loss: If the body loses a large volume of blood, the brain detects the fact that it's not getting enough oxygen (decreased blood) and automatically dilates the vascular system and increases the heart rate. If the blood loss is due to an

open wound, this serves only to lose blood faster! So when we talk about EDA's lack of reliance on central control, we mean that the control is distributed in the form of business rules—and distributed rules must be configured to trigger appropriate actions. The event components contain business rules that are implemented as each event component is activated. The result is an application, or set of applications, that operates under control, but not with a central controller.

Event-driven architectures insert context into the process, which is missing in the central controller model. This is where the potential for a truly dynamic system emerges. Processing information has a contextual element often only available outside of the central controller's view. Even if that contextual change is small, it can still have bearing on the way data should flow.

One contextual stimulus is the Internet. The Internet has opened up businesses to a new undressing. Business-to-business transactions, blogs, outsourcing, trading partner networks, and user communities have all cracked open the hard exterior of corporations. They provide an easily accessible glimpse into a corporation's inner workings that wasn't present before. This glimpse inside will only get larger with time making the inner workings public knowledge and making media-spin-doctoring of unethical practices more evident.

Don Tapscott in *The Naked Corporation*[2] talks about how the Internet will bring moral values to the forefront as unethical practices become more difficult to cover and financial ramifications increase. Businesses will be valued on their financial standing along with reputation, reliability, and integrity. This means businesses will have to change their process flow based on external conditions such as worldly events and do so efficiently.

Information is being aggregated in different ways. Business processes are changing and being combined in real time with external data such as current worldly events. Because of the increased exposure through the Internet, questionable businesses practices are being uncovered. Sometimes, these practices are unknown to the core business, hence businesses want to react quickly to the publicity. Imagine a news investigation that uncovers a major firm is outsourcing labor to a company involved in child slavery. For example, company X is exposed for buying from a cocoa farm in West Africa's Ivory Coast that uses child slavery. The business would immediately want to stop their business transactions with that company and reroute them to a reputable supplier before the damage becomes too great.

For ethical reasons, eBay continually blocks auctions that attempt to profit from horrific catastrophes like major hurricanes, a space shuttle accident, or even a terrorist attack like 9-11. Imagine the public impression of eBay if this was not practiced and they profited from these events.

Now imagine having a system that's worldly aware enough to circumvent business processes if these cases should occur. Suppose this system had an autonomous component that compares news metadata with business process metadata and curtails the process at the first sign of concern. The huge benefits definitely outweigh the calculated risks. Simply rerouting a purchase order to another supplier with comparable service levels definitely has a big upside. If the autonomous deduction was correct, it might have saved the company millions in bad press while maintaining their social responsibility. If it was wrong, then no real harm was done because the alternate company will still deliver on time.

A similar scenario could support eBay's ethics. An autonomous component that compares news metadata with auction metadata could withhold auctions based on real-time news events. If correct, it could save the company from public embarrassment. If wrong, little harm was done other than to delay an auction start time.

EDA can provide this dynamic monitoring, curtailing, and self-healing. Event-driven architecture facilitates bringing these external contexts into the business process. The idea is that the separation between concrete business process and day-to-day reality is blurring. Businesses might be required to change their process based on unexpected external events. This is much different from the days where an end-to-end business process happened within a company's boundary (and control). Combining this need with the traditional business need for rapid change means flexible architecture design is paramount. One way to ensure this flexibility is through the SOA/EDA way—by reducing central control and adding context to the business process.

BAM—A Related Concept

Business Activity Monitoring (BAM) is related to EDA, but different enough that we discuss it in brief. Our goal is to help you differentiate between BAM and EDA, as the two ideas are often used interchangeably in IT discussions. We do not think they are interchangeable.

BAM is the idea that business decisions would be better and more timely if they were based on timely information extracted through business activities that are exposed near real time. Too often, decisions are made based on warehoused data that is stale or misrepresented because of the available gathering technique. Event-driven architectures make it easier to tap into key business activities. BAM components monitor these activities, aggregate the information, watch for anomalies, send warnings, and represent the data graphically.

Historically, most of the activity in this area was achieved with in-house built dashboards. Now we're seeing more vendor products in the space. BAM is most useful in situations where quick critical decisions are important. Interesting applications of this concept include illustrating Key Performance Indicators (KPI), watching for homeland security anomalies, monitoring supply chain activities, and discovering business-to-business (B2B) exchange patterns. Implementing a BAM solution within your EDA is almost always a good idea.

Chapter Summary

- In this chapter, we move forward with our metaphor of EDA as the enterprise nervous system and match the EDA components—event producers, listeners, processors, and reactors—to their equivalent in the nervous system. Event producers and consumers are likened to the sensory nerve endings that pick up and relay information about our senses to our brain, which is like an event processor. Reactions, such as physical movements, are like the event reactors. For additional context and framework, we look at event-driven programming, a core technology of most PCs, as a comparable example of events, event listening, and event processing on a lower level of functioning than an EDA.
- To complete our understanding of how EDA works, we then carry this enterprise nervous system idea further and take an in-depth look at the characteristics of EDAs and their components. Again, our focus is on the EDA of the future: an implicit, complex, and dynamic EDA, one that can adapt easily to changes and continually expand its reach of event detection and event reaction.

- EDA components must be loosely coupled to function dynamically. Loose coupling requires that EDA components have low levels of preconception about each other and maintainability. An EDA works best if each component functions independently, with little need to know about the other components it is communicating with, and few ramifications if one component is modified.
- EDAs, unlike conventional applications, do not rely on central controllers.
- Events (state change notifications) are central to an EDA. An event can take the form of a message and an EDA is a message-based idea. To work, an EDA's loosely coupled components must be able to produce and consume messages. The messages could be related to event listening, processing, or reactions. The more easily the messages can flow across the EDA (which might span multiple enterprises), the better the EDA will work.
- Asynchronous, or publish/subscribe (pub/sub) messaging, is one of the best foundations for an EDA. As the EDA components communicate with one another, they feed messages (events) into an event bus. Event listeners receive the events, and then EDA components process the event data as required by the EDA's designed purpose. Pub/sub is ideal for EDAs because it removes a lot of message flow dependencies from individual components. It is simpler, for example, to connect event listeners using pub/sub than to tightly couple them together, where changes in configuration are costly and slow to accomplish.
- To achieve loose coupling and asynchronous messaging, an EDA relies on message intermediaries. In some cases, these are known as service buses.
- The ideal EDA, therefore, is a loosely coupled, pub/sub-based architecture, with low levels of preconception and high degrees of maintainability among the components.

Endnotes

1. Wikipedia. Event-Driven Programming. August 2004. http://en.wikipedia.org/wiki/Event-driven_programming.
2. Tapscott, Don. *The Naked Corporation*. New York: Free Press, 2003.

Index

A

B

C

D

E

F

G–H

I

J–L

M

N–O

P

Q–R

S

T

U–V

W–Z