Event-Driven Architecture Complements SOA

Event-driven architecture and service-oriented architecture are compatible but distinct concepts, each with its own advantages and limitations. Enterprises need both.

As mainstream enterprises begin to adopt service-oriented architecture (SOA) on a broad scale, they are beginning to see more need for another design approach — event-driven architecture (EDA). SOA and EDA have many things in common, because they both are ways of combining multiple software modules into large distributed applications. However, they differ in the way they organize the relationships among the modules, and this makes them appropriate for different purposes. At the risk of oversimplifying, SOA uses directed, generally bidirectional, request/response communication to delegate work to a subordinate procedure, whereas EDA uses unidirectional messaging to communicate among two or more, largely independent peer procedures. Applications often intermingle SOA and EDA constructs.

Basic SOA:

- Has one-to-one connections
- Has flow routing that is directed by the client (sender)
- Employs a linear path of execution through a hierarchy of modules
- Is closed to new, unforeseen input once a flow is started

Basic EDA:

- Supports many-to-many connections
- Has a flow of control that is determined by the recipient based on the message itself
- Supports dynamic, parallel, asynchronous flows through a network of modules
- Can react to new, external input that may arrive at unpredictable times

Core Topic
Application Integration and Middleware: Application Integration

Key Issue
What will be the core technologies and critical success factors for implementing a flexible enterprise nervous system that can accommodate ongoing growth and modifications?
Note that many people think of EDA as a kind of embellished SOA, based on a loose definition of the word "service." They consider any distributed application program to be a service if it can process data from disparate senders (which generally implies that the program's interface is described in a metadata repository that is accessible over a network). However, this ignores a very important distinction. An SOA service acts as a server in the classical software engineering client/server sense — that is, it is client-driven and uses a "find-bind-invoke" relationship. By contrast, an event-driven module is not technically a "service," because it has no binding to any application client and does not act in a server role. Regardless, people commonly speak of an "event-driven service," although it is an oxymoron. A module is either client-driven or event-driven — it cannot be both at the same time.

Other people say that any application that is implemented using Web services is an "SOA." However, this confuses two concepts: conceptual design and implementation. SOA and EDA are conceptual design patterns that can be implemented with many different kinds of middleware. Web services are programs that adhere to certain middleware standards — specifically, a particular program-to-program communication protocol (Simple Object Access Protocol, or SOAP) or a particular documentation mechanism (Web Services Description Language, or WSDL). EDA and SOA applications can be implemented with Web services, but neither requires Web services. Because Web services evolved from Extensible Markup Language (XML) remote procedure call (RPC), they apply to SOA better than EDA today, but we expect that Web services standards will be enhanced to cover more aspects of events in the future.

The following elaborates on the nature of SOA and EDA.

**Similarities**

*Modularity*

SOA and EDA are design patterns for distributed systems — that is, applications that encompass multiple programs that can run on different computers connected by a network (in a simple case, the modules may run on the same computer). Each module is a business component in the sense that it is a program or set of programs that implements a procedure that is meaningful in business terms, such as "look up customer account balance" or "update product price." A business component does not have to be a Component Object Model (COM), Common Object Request Broker Architecture (CORBA), JavaBean or .NET component, although it can be. Components are modular and encapsulated. Modularity means dividing big problems into smaller problems.
Encapsulation means hiding the data and logic in each module from uncontrolled external access (hence, the metaphor of a "black box" is often applied). Encapsulation shields external developers from having to understand the internals of a module or to keep up-to-date with changes that are made inside a module. In an SOA, a module is a server or "service implementation," and the program that invokes that service is the "consumer" or "client." EDA modules or components are event emitters ("sources"), recipients ("sinks") or both.

**Connections**

Program-to-program communication is at the core of both SOA and EDA. SOA modules communicate with each other through documented contracts called "interfaces." SOA developers document their interface definitions (for example, using WSDL) to help other developers use their services. Similarly, EDA modules have defined process interfaces and are able to communicate because the developers of the source and sink modules understand a common event descriptor. Most new event-driven applications will use XML Schema Definition (XSD) to document the events, and eventually WSDL will probably acquire more features aimed at events.

Changing an interface usually causes a lot of work. A change to an SOA interface requires a change in:

- The client program
- The client interface proxy (a software stub that marshals the data and communicates with the server stub using SOAP or another protocol)
- The service interface stub (which demarshals the data and invokes the service implementation)
- The service implementation (a distributed application program)

Similarly, an EDA relationship is determined by its event descriptor, so most changes in the event descriptor will require modifying:

- The source
- The sink
- The associated software libraries or adapters used for emitting or receiving the events

In theory, minor changes could be made to one side of an SOA or EDA connection without changing the other side, as long as
the application logic is not disrupted. For example, a source or client could start sending an extra data element that is ignored by the receiving sink or server. However, most changes to an application will logically require changing both sides of the connection to meet the evolving business needs. Moreover, in many cases, the nature of the middleware used to implement the SOA or EDA dictates that both sides must be changed at the same time even if the logic of the business solution does not require it. If a change is to be made to one side but not the other, the disparities in the contract are often resolved by inserting an integration broker, gateway or other software intermediary to transform the interface or events.

Differences

Relationship Cardinality

Each runtime connection in a simple SOA is a 1-to-1 relationship: one client (for example, a Web services consumer) invokes one service (for example, a Web services provider). By contrast, a runtime EDA relationship can be many-to-many (m:n). The first part of this means that one event message may be delivered to zero, one or thousands of sinks (a 1:n relationship, see Figure 1). EDA is more efficient than SOA if there are multiple destinations for the same data, because the source sends the event only once, whereas an SOA client would have to make successive calls (each through a separate interface) to achieve the same wide distribution. Because EDA events can be delivered to multiple destinations, the flow of control can fork into parallel streams of execution. By contrast, the flow of execution in a simple SOA is a linear path. The second part of this is that each event sink may receive multiple events before taking any action, enacting a many-to-1 (m:1) relationship that joins the flow of control (hence, the overall relationship in an EDA can be m:n). A simple SOA pattern does not support forks or joins, although developers can extend SOA by writing multithreaded applications or using an integration broker or business process management (BPM) engine to configure a more complex flow that includes statically defined forks and joins.
An SOA application is hierarchical in the sense that clients can delegate work to one or more services and, in turn, each of those services may delegate work to other services. An SOA client holds state information, but SOA servers are usually designed to be stateless so they can readily service other clients. The client directs the flow of control by specifying the service to be invoked (for example, “look up customer account balance”). Of course, the client need not know the specific IP address or even which enterprise is hosting the service, because these can be determined at runtime — for example, the middleware may do a look-up using a uniform resource identifier (URI). However, the client — a third-generation language (3GL) application, script, servlet or BPM engine — always decides when and whether to invoke a particular service.

Almost all SOA calls have a response back to the client, but that is not a requirement. If there is no response, it is still an SOA as long as the connection is 1-to-1 and the client specifies the action to be taken. An SOA response may be synchronous (the client blocks — that is, stops working while waiting for the response) or asynchronous (the client performs work while waiting for a deferred response). Asynchronous SOA is more efficient if the client is able to do some work while it waits. Asynchronous SOA is also more resilient, because temporary problems in the network or the service implementation do not disrupt the client. As long as the response is received within a specified time, the
client runs properly. Therefore, asynchronous SOA is usually better than a synchronous SOA if the modules are managed by disparate groups separated by a WAN. However, synchronous SOA can be easier to program and may be appropriate if the servers are reliable and the application logically needs a response before it can proceed.

**Event Flow**

An event-driven application is conceptually a network in which multiple, simultaneous streams of execution may run independently of each other. A sink can listen to thousands of incoming events from many different sources. The flow of control is determined by the event sinks, and it depends on the nature or contents of the event. Data that represents the state of the business process instance is contained in the event message. In a simple event-driven application, the sink is an application module. In other cases, the sink is an integration broker, BPM engine or, in complex-event processing (CEP), an event manager.

Developers do not build any assumptions about the event sinks into the source modules, and there is no reply back to the original sender. Developers of the source module may think they know what logically has to happen next in the business process, but there is nothing in the source module that ties it to a particular sink. Source modules merely send the event into the opaque middleware infrastructure. If business requirements change, event sinks may be added, modified or deleted without affecting the source or any other sink (if the event definition is not varied). By contrast, an SOA client must be explicitly modified to add or drop an SOA service (for example, a simple call statement always invokes one server, not zero or two).

**Serve Different Purposes**

Developers use SOA when the nature of the business problem requires a request/response relationship. If a client module needs some answer (immediate or deferred) from a subordinate procedure before it can complete its work, SOA is appropriate. SOA is more flexible than traditional application design approaches because, as long as the interface is unchanged, SOA modules can be individually modified or replaced. However, basic (non-BPM-enabled) SOA is conventional in its reliance on a statically defined, single path of flow.

EDA meets different needs. If a business problem can be solved using one-way event delivery, events are preferable to SOA because of the greater independence of the senders and receivers. EDA not only helps to organize the processing flow (as
an SOA does), but it also improves the understanding of what is happening in the application. Events enable applications in which:

- Multiple processing streams may execute simultaneously.
- The timing of events, such as the beginning or end of a step, or the arrival of additional external input, is unpredictable.
- There is a need to dynamically add, drop or modify processing steps without changing any running modules in any way.

EDA is decoupled; asynchronous SOA is loosely coupled; synchronous SOA is somewhat coupled; and monolithic (non-SOA) applications are generally tightly coupled.

EDA and SOA Are Complementary

EDA is entirely compatible with SOA. An EDA source can be an SOA client or server application, and an EDA sink can be an SOA client program. Integration brokers and BPM facilities will often be used to coordinate the combined SOA/EDA applications (see "The Case for Event-Driven Design").

**Bottom Line:** Event-driven architecture and service-oriented architecture have many similarities. Both support distributed applications that go beyond conventional architectures, both use a modular design based on reusable business components, and both may be enabled through Web services. Architects and developers must understand the local business requirements and process models to determine whether SOA, EDA or some combination of them is right for each aspect of each new business process.