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.

- 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.