EVENT-DRIVEN ARCHITECTURE

How SOA Enables the Real-time Enterprise

Hugh Taylor | Angela Yochem | Les Phillips | Frank Martinez

Event-Driven Architecture

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.

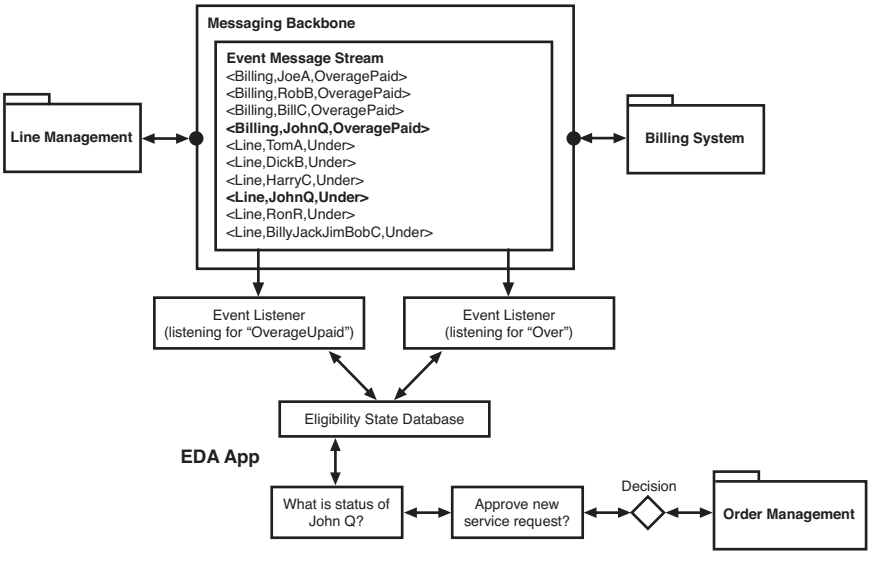

Figure 3.13 The EDA-based application subscribes to event data that is published and consumed by event listeners on separate systems. This gives the EDA-based application the ability to have awareness of changes in state related to John Q without tightly coupling any of the applications involved in the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