Alternative Technologies

Enterprise Integrity: BPMS Concepts Part 8

Vol. 3, No. 8

If you have a BPMS, or at least a BPM solution, your business is positioned at the leading edge. I hope that this series has convinced you that a BPMS is a worthwhile pursuit, whether you buy or try to build it. Either way, it is unlikely that you will find exactly what you need, and very likely you will want to extend and integrate the functionality. You can't do that well if you don't understand process-oriented programming. This month I discuss one remaining, but important topic in this series: design and development in a BPMS environment.

In the ideal BPMS environment, every software resource is completely process enabled. While not impossible to use ordinary software-implemented business functions or activities, traditional software does not enable full BPMS potential. Completely process-enabled software must contain no implementation of a business rule. Thus it is either (a) an implementation of an atomic service, or (b) purely declarative (no procedural code). Object-oriented programmers have to be taught to add process-oriented discipline to their habits. They must learn to avoid embedding business rules or process in the code, componentize around atomic services, publish a complete service contract, and event-enable each service invocation.

Process-oriented software consists of a set of componentized services in a service-oriented architecture. Which services are invoked and in which order is controlled by the BPMS. BPMS solves a key problem with service-oriented architectures: It alleviates the tendency toward hard-coded and uncoordinated service invocation. The components have an obligation to publish their behavior to a resource manager, which optimizes requests by sending them to a capable component. Component behavior must be more complete than is typical in object-oriented programming. Quality of service, transactional requirements, time-outs, results delivery mechanism, synchronous or asynchronous interaction, security requirements, and resource requirements specify environment and behavior as part of each component's service contract.

At the lowest level of decomposition, components are written to a virtual machine, hiding all details of the available hardware resources. The resource manager optimizes the use of hardware resources according to their operational characteristics (as registered in a configuration repository), meeting declarative requirements and goals. For example, consider sending a hardcopy document to someone. Rather than selecting a specific fax number or network printer, the document is simply printed with the optimization goal being time-to-delivery and required quality of hardcopy. The resource manager then selects a method of

www.AlternativeTech.com mcgoveran@AlternativeTech.com

printing that meets requirements while minimizing time-to-delivery, probably an available fax located closest to the intended recipient. Hopefully you get the idea of this complicated topic: We create a kind of *dynamic* polymorphism.

At a higher level, atomic business actions or services are implemented. These business components must contain no references to, or implicit assumptions about, the hardware resources on which they are ultimately executed. Atomicity is a crucial concept, meaning that every internal constituent is essential to delivery of the specific business service, and no independent semantic content to the business. Process-oriented programming involves constructing abstract business services from such atomic business actions. Abstract business services never contain business logic; any logic is due solely to choice of algorithm. Business logic is referenced within the component symbolically as business rules, with the actual rules maintained externally via a rules engine and database. This clean separation of business logic from implementation logic improves maintenance and flexibility, encourages business knowledge capture, and enables knowledge management.

Process-enabled components are invoked by declaring an event (similar to raising an exception), and never directly. The event listener may be implemented either directly within the component, or indirectly via an event broker. The event broker, in essence, converts events into strongly typed messages that it publishes to components that have subscribed to events of a type. It may also invoke the component, passing it the typed message as a parameter. The advantage of an event broker is that it can listen for many events and permits many components to register for those events.

Hopefully this brief introduction to key process-oriented programming concepts will suggest some new possibilities to you. The many business and technology implications of using such an approach to design and development cannot be fully explored here, so I encourage you to consider them on your own. Re-read the entire BPMS Concepts series and share it with your business managers as well as software developers. Who knows, your IT staff just might be motivated to contribute to your company's *enterprise integrity*.

.