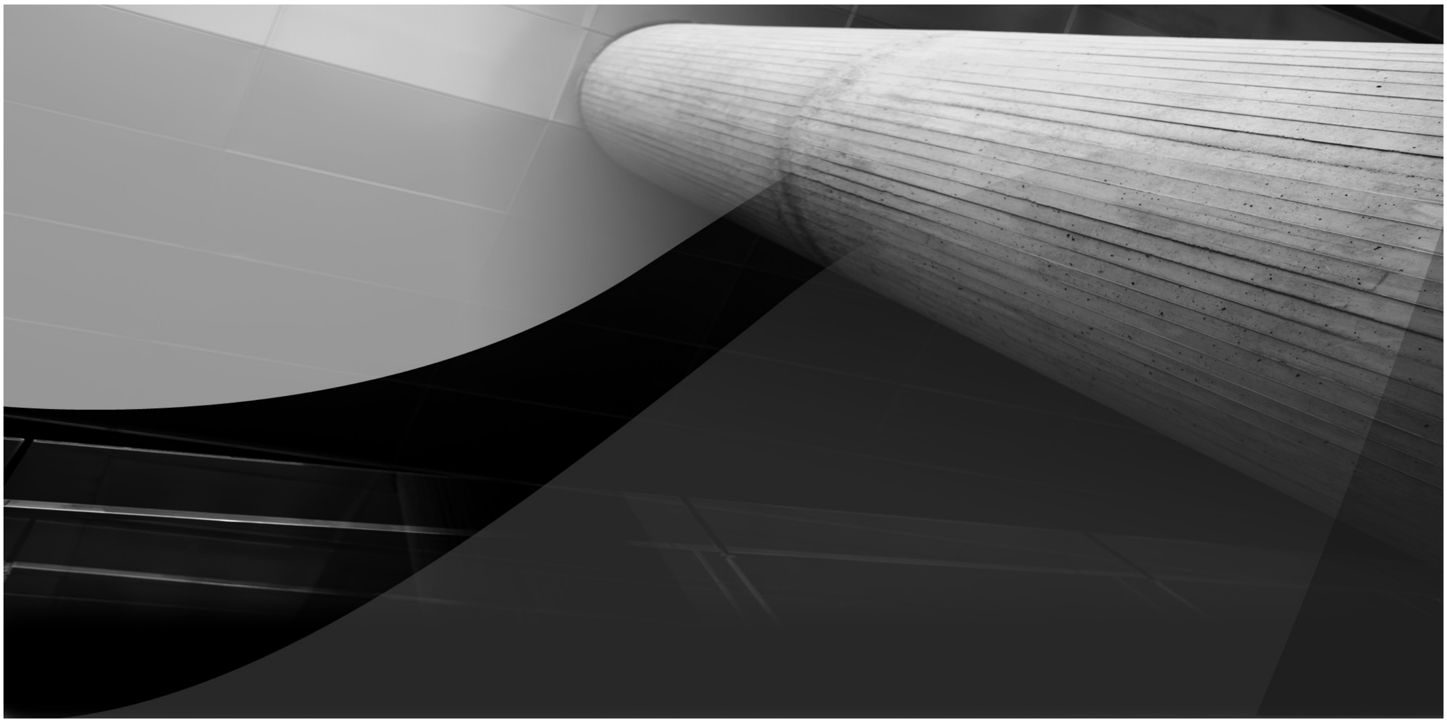


CHAPTER 9

Event-Driven Architecture for Super Decoupling

SOA is about decoupling and reuse, leading to business agility. In the previous chapters we have seen many examples of decoupling, both within and between our SOA composite applications, as well as between these composites and external services and systems.

The use of XML and Web Service standards is good for decoupled interoperability across heterogeneous technology stacks—for example, file systems, databases, Java applications, packaged applications, and SOA Suite. The integration between service components based on the WSDL contracts and the SCA infrastructure allows us to use the best tool for the job—Business Rules for business logic, Mediator for routing and transformation, BPEL for stateful processes, and technology adapters to leverage functionality in other platforms.

The asynchronous capabilities that queues such JMS and AQ, as well as the events introduced in this chapter, provide us with also allow for temporal decoupling where consumer and provider can communicate without having to be available at the same time.

In terms of decoupling, we have at least one other challenge left: How do we make sure that services are called at the right time? Some services provide clear value to their invoker, such as the patient data or the result of a calculation. Such two-way services will be called whenever their functionality is desired by an application. The application is functionally decoupled from the service—the canonical model and reusable Web Service contract in combination, possibly with service endpoint virtualization through the Oracle Service Bus (see Chapter 13) or a service registry, take care of that. However, the application still needs to explicitly invoke the service, needs to know some endpoint location, and work according to the service contract.

The story is even more interesting for one-way services, which may need to get into gear for processing the newly received appointment request, for absorbing the change of address for a patient, for dealing with the sudden unavailability of an operating room as a result of equipment failure, or for handling the patients now 30 days late in paying their bills. Who is responsible for calling these services? No one will call them to get something out of them—because they do not return a response. Other services and applications may have or even generate the information that the one-way services need to get. But whose responsibility is it to get it to them? How should these information owners know which one-way services are interested in their data? And should the onus be on them to explicitly call these services? Surely we do not want to modify and redeploy applications whenever a new consumer of their information comes along—or an existing one loses interest. That would not be decoupling at all!

This chapter introduces the Event Delivery Network (EDN) in SOA Suite—a facility that provides advanced decoupling by mediating events between producers and consumers that are unaware of each other. Composite applications can subscribe to one or more of the centrally defined business event types and are notified by the EDN whenever an instance is published of one of those types. More specifically, both Mediator and BPEL components can produce and consume events. Events can also be correlated into running composite instances through BPEL components.

The online chapter complement contains some of the XML snippets and other sources for this chapter, as well as screenshots and detailed step-by-step instructions to follow through the examples described in the chapter.

Event-Driven Architecture for Super Decoupling

Event-Driven Architecture (with the obvious acronym of EDA) made a lot of heads turn. Seen as the successor to SOA by some and as a welcome complement to SOA by others, EDA is clearly on to something. Extremely Decoupled Architecture would be a perfect secondary meaning of the acronym—because extreme decoupling is one of the things that EDA adds to "traditional" SOA. In the world of EDA, events are the messages, replacing direct service calls. Events are not targeted to a specific service provider. Events are published with some logical name on some central, generic infrastructure that the event producers use in a fire-and-forget mode. After publishing their event, these producers have no more responsibility for it. They have done their job by handing the event to the central event-coordinating facility and no longer have strings attached to the event. In fact, they may later on even consume their own event just like any other consumer—because their origin is unknown to event consumers. Events typically have a header and a payload. The header contains metadata such as the event type and a timestamp. The body or payload contains the details that describe the facts of the event instance that business logic in the consumer will process.

Anyone interested in occurrences of a specific type of event can subscribe to it, registering their interest in that event type with a central facility that coordinates events. This coordinating facility receives all events that get published, relieving the event publisher of the responsibility for the event. After receiving an event, it will traverse all subscriptions for the particular event type, taking into account any filters that may have been defined on such subscriptions to see whether the specific event occurrence should actually be forwarded to the subscription's consumer, and propagate the event to all qualifying consumers. The event coordinator may support mechanisms to retry delivery of an event upon initial failure or deferred delivery for currently unavailable consumers. Figure 9-1 shows an example of an EDA environment with two types of events: The light and dark cubes are published to the event coordinator and propagated to consumers that have subscribed (dashed lines) to the event type(s) of their choice.

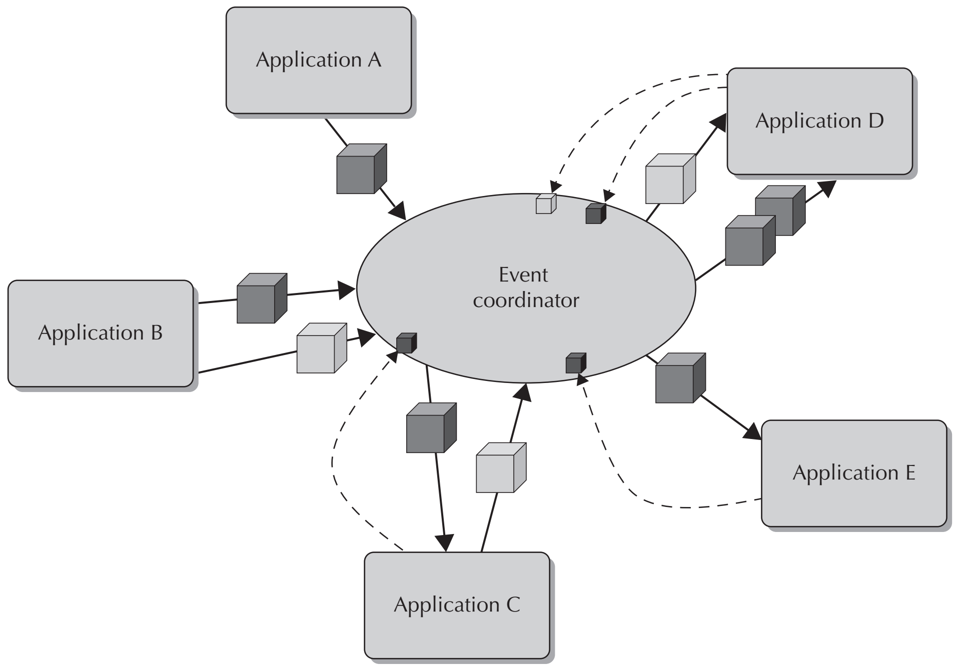


FIGURE 9-1. *The fundamentals of Event-Driven Architecture*

NOTE
None of the producers of events are aware of these registrations. They should continue to publish their events even if no subscriptions exist at all. They should neither know nor care.

The terms *SOA 2.0* and *Event-Driven SOA* have made some inroads into the SOA community. They both indicate a service-oriented architecture where event mechanisms are used to further decouple applications and services from each other. Instead of coupling applications and services that are sources of business events to the (often one-way) services that have a need for the information, events are used to convey the data through a generic facilitating medium: an event coordinator. In the SOA Suite, the role of event coordinator is implemented by the Event Delivery Network.

Introducing the Event Delivery Network

Oracle SOA Suite 11*g* comes with the Event Delivery Network (EDN), an infrastructure that provides a declarative way of defining, publishing, and registering for consumption of business events. The EDN enables implementation of the EDA patterns in the SOA Suite.

The Event Delivery Network is a man-in-the-middle, a central coordinator that interacts with three types of entities: publishers of events, consumers of events, and the events themselves. The publishers—composite SOA applications or external parties such as Java applications or PL/SQL code running in the database—create an event and publish it by telling the man-in-the-middle about it.

However, before events can be published, their meta-definition needs to be in place, consisting of a (fully qualified) name and an XSD definition of their payload. The payload of an event is the data associated with it, provided by the publisher and available to the consumer. The meta-definitions of the business events are defined in EDL—the Event Definition Language. EDLs are deployed inside composites or stored in MDS. These event-definition files typically import one or more XSD documents that provide the element definitions on which the event payload is based.

EDL is just another XML language—based itself on an XSD (edl.xsd in the JAR file bpm-ide-common.jar) that is registered with JDeveloper. Note that there are no explicit references to EDL files, not from composite.xml or from any of the components' definitions files. All EDL files in a project help provide events that the components can subscribe to or publish, and all EDL files deployed to an SOA Suite instance are available to all composites in that SOA Suite. One EDL file can contain multiple definitions of event types.

The Event Delivery Network works across and beyond the SOA Suite, coordinating events from and to all composites running in the SCA container. The event definitions should therefore ideally be generic, based on canonical data model definitions.

Once event definitions have been registered with the EDN, subscriptions can be created on those events. Composite applications register their interest in events of a specific type with the EDN through Mediator or BPEL components that consume such events. BPMN components—which will be introduced in Chapter 11—can also subscribe to EDN events that are consumed as BPMN signal events.

Upon deployment of an application with event-consuming components, the subscriptions are automatically detected by the EDN and used to distribute the published instances of those events. Publishers of events do not have to be registered beforehand; anyone can publish an event of a type that is defined through EDL.

As you probably already understand, events of a specific type can be published by many different publishers, both inside the SCA container and outside of it. Primary publishers of events to the Event Delivery Network are Mediator, BPMN, and BPEL service components. The SOA

From E-mail to Twitter—Decoupling Through Publishing

Jenny is not necessarily a nosy person. She just happens to know a lot about what is going on in her corner of St. Matthews. New nurses and doctors, staff calling in sick, rare new cases, VIP patients, extra-special operations, and dates between staff members—Jenny knows it all. And her colleagues know that she does.

One day, Victor, one of the interns, asked Jenny to let him know whenever an emergency operation would be scheduled. And Jenny was happy to oblige; whenever the schedule was overhauled because an emergency operation had to be performed, Jenny would page the intern.

Word got around, and after a few weeks, other interns came to her with the same request. And Jenny was a good sport and added their names to her list of people to inform upon interjected operations. Then things started to get a little trickier when she was asked by one or two desperate single nurses to keep them informed of newly admitted, apparently single male patients—via e-mail this time, because paging would be somewhat ridiculous.

Victor came back to her and—eager to participate in more operations than he was lined up for—talked her into letting him know whenever one of his colleagues called in sick and was scheduled to scrub in. And if she could, please page him as well as leave a voicemail on his telephone—because he might be at home or en route.

Jenny could not cope anymore. Keeping track of all the people, the information they wanted, and the channel through which they requested to get it just became too much. She had her own job to do as well!

Then she found a perfect solution: Instead of maintaining lists of people's interests and calling, paging, or e-mailing them whenever a nugget of information fitting with their particular request had become available, she started to use the corporate Twitter. Every tidbit of information that came across her desk she turned into a tweet. And she told everyone who wanted information from her to just "read the feed!"

Life became so much easier. She tweeted her news flashes, not knowing nor caring by this time whether anyone would read them. New readers could join in, and old ones could vanish from the crowd—temporarily or permanently. It no longer affected her.

At some point there was a request to somehow filter her tweets—in order to distinguish between operation warnings, sickness notifications, and hunk alerts. She started to add hash tags (#opr, #sck, #hnk) to her tweets and thus made everybody happy.

Her counterparts in other departments joined in and started to tweet their news as well, in a similar vein. Victor at some point happily scrubbed in on an operation in a remote part of the hospital, thanks to the alert some other "Jenny" had twittered.

Unknowingly, Jenny migrated from a distinctly coupled communication pattern to a much more efficient, decoupled approach based on publishing news and messages in general (to the ether) rather than sending them to individual recipients. With a much lighter responsibility and workload, she can make even more people happy than she realizes.

Suite also exposes APIs in Java and PL/SQL for publishing events. ADF Business Components can be configured to publish events to the EDN when data manipulations on entity objects occur (see Chapter 20). The FMW Control has a facility for publishing events for test purposes—we will be using that feature later in this chapter.

Many different composite applications can subscribe to an event. Each of these applications will receive notification of the events from the Event Delivery Network. Figure 9-2 shows the Event

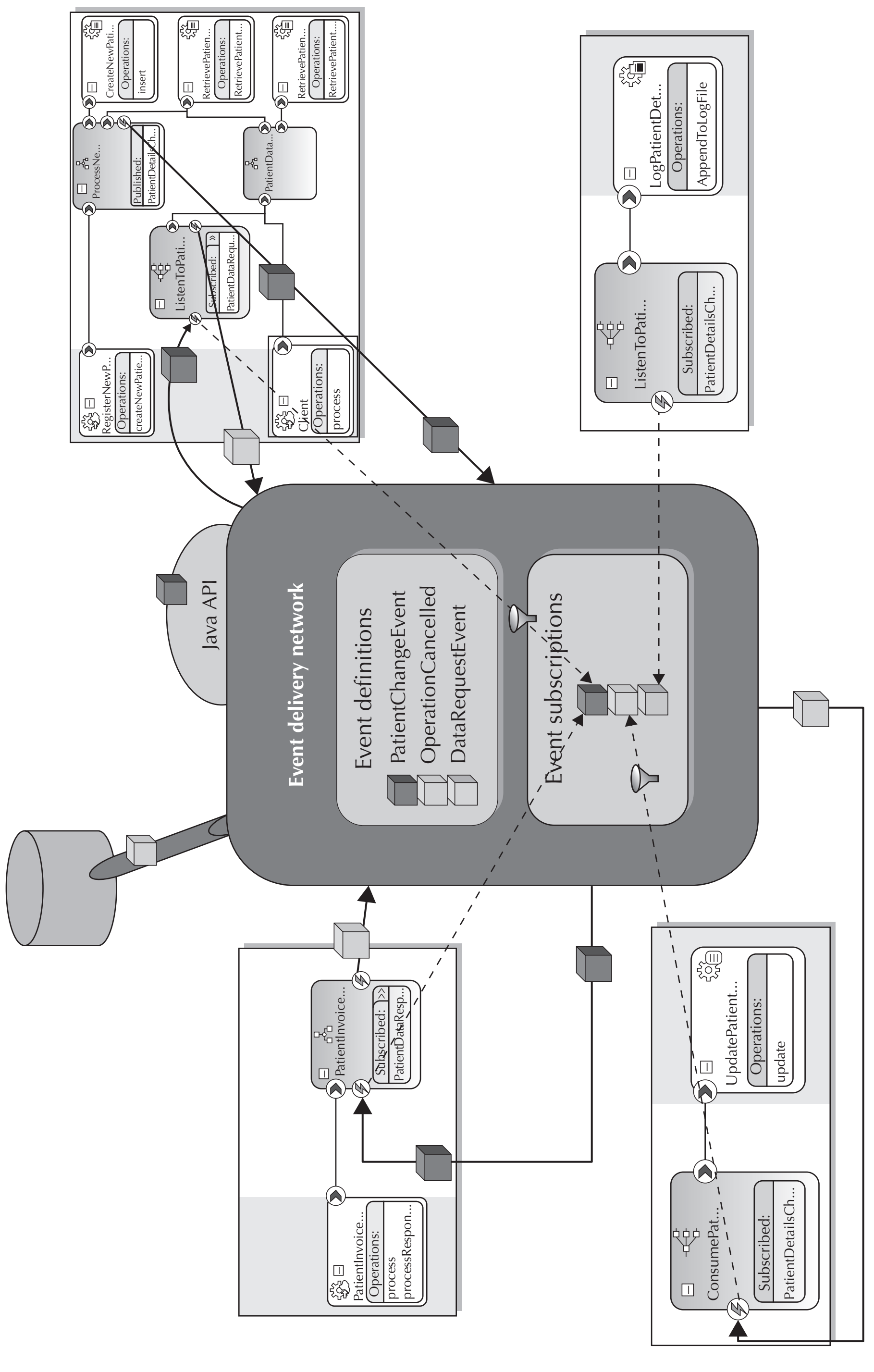


FIGURE 9-2. *The Event Delivery Network*

Delivery Network working with three business event types and four composite applications that each have one event subscription (red dashed lines). Two of those subscriptions have an explicit filter that sifts the relevant instances of the red and yellow events. Two publishers broadcast the red event (PatientChangeEvent) and three broadcast the yellow (OperationCancelled) event. The thin blue lines show red and yellow events being passed to consumers based on their subscriptions.

First Round with EDN: Consuming Events

St. Matthews is moving toward the Event-Driven Architecture utopia. Everyone involved with the SOA initiatives agrees that events are a perfect means of propagating information through the entire organization, making it available to everyone yet forcing it on no one.

A successful adoption of the Event-Driven Architecture can be achieved in steps. The hospital can start with defining one or a few business events and ensure that all communication around those events is implemented in an event-driven way. This requires the right mindset *and* discipline. All owners of processes and applications where these events are generated must be identified and convinced to explicitly publish those events on the Event Delivery Network. All services and processes that need to be triggered by those events must be adopted to consume them through a subscription with the EDN—rather than waiting to be invoked by someone with the information. Mindset and discipline—the latter to take on the responsibilities that come with EDA. There is a responsibility to push events by the instigators or first points of contact in the organization (for example, the applications that register cancellations, registrations, orders, and modifications). And applications, processes, or services that run on information available through events have a responsibility to pull the information from the EDN via subscriptions on the events.

We will dive into the implementation of EDA using the SOA Suite Event Delivery Network and the case of the patient's address changing as our first example.

Synchronizing Patient Data Using the Event Delivery Network

One of the major complaints from patients about St. Matthews is the poor handling of changes in their personal data. The number of times bills and other mail has been sent to the old address of a patient—leading to late payments or even actions by collection agencies that turned out to be rather unjustified—is way too high. Or simply the failure to change the surname after a divorce or marriage or the title after sex-reassignment surgery—even when performed at St. Matthews itself. Patients frequently are overheard sighing. "I *did* inform the hospital! I notified *them* weeks in advance! Why can't *they* understand a simple thing like a family moving!"

Patients have no clue—and should not need to know—that behind the façade with the big sign "St. Matthews Hospital" is a plethora of departments, computer systems, processes, administrations, and procedures that do not necessarily form a unified entity where all speak with one voice and listen with a single ear. "Informing 'the hospital' of some fact and expecting that everyone in the hospital thereby becomes aware of it is plain naïve!" seems the general opinion of the hospital staff, although it is not something one would say out loud to a patient, of course.

Consuming Patient Change Events

Because of the annoyance felt by the patients, and the staff to some extent, as well as for efficiency reasons—the time spent on correcting the errors just described is substantial—it has been decided that changes in the patients' contact details are among the first to be tackled through events. We will see how Frank implements a simple composite application that subscribes to the Patient Details Change event and uses the payload to synchronize his patients database. He uses a Mediator to consume the events and forward them to a database adapter that updates the patients table based on the event payload. Later on he will expand the composite so that the customer change is propagated to all systems within St. Matthews that store and use patient information.

Enforcing a regime where any change in patient details is turned into an event on the EDN, wherever that change first enters the hospital, is beyond the scope of this chapter, as are a discussion on how the correctness of the information is verified, when exactly the new data first becomes valid, and any other security issues that need to be dealt with when using events. However, note that events are not subject to authorization mechanisms: Any composite can register for any event type on the EDN and every instance of events is delivered to every consumer—when allowed by the optional filter expression on the subscription. No authorization can be specified or enforced.

We will also avoid a discussion on whether a missing element should be interpreted as a value that was not changed or one that was nullified. We will focus only on the implementation in the SOA Suite of the simplest interpretation of the event.

The steps are fairly simple then:

1. Create a new SOA composite application called SynchronizePatientsInformation, with a project carrying that same name. The application will initially consist of an empty composite.
2. To keep things simple in this example, just create a new XSD document called PatientDetailsChangeEvent.xsd with its target namespace set to http://stmatthews.hospital.com/events. Create a PatientDetailsChangeEvent and a complexType called PatientDetailsChangeEventType that defines the payload for the event—a timestamp and a patientDetails element based on the patientDetailsType in the imported schema document CanonicalPatient.xsd that contains a simple representation of a patient with her contact details (the same that was used in previous chapters; see Figure 9-3).

   The PatientDetailsChangeEvent is defined in an EDL file—as a hospital-wide event type with a canonical payload—not specific to any application or service and based on elements defined in the canonical data model. Note that the EDL file has to be created and deployed as part of an SOA composite application, even though it does not really belong to any one application—it is applicable throughout the SOA Suite container and is more or less "community property." In Chapter 18, we will see how MDS can be used to centrally store, share, and manage EDL definitions.
3. Open the composite.xml file in the editor. We will now create the EDL file, as shown in Figure 9-4. Bring up the Create Event Definition File window by clicking the event icon in the upper-left corner. Enter **PatientEvents** as name for the EDL file. Accept the default derived namespace. Click the green plus icon to add the first event definition. Select the element PatientDetailsChangeEvent from the XSD with that name in the Type Chooser pop-up window. Enter **PatientDetailsChangeEvent** as the name for the event.
4. Create a Mediator called **ConsumePatientDetailsChangeEvents** that subscribes to the PatientDetailsChangeEvent (see Figure 9-5). Drag the Mediator component from the palette to the composite. Specify the name and select the template Subscribe To Events. In the Event Chooser, select the PatientDetailsChangeEvent from the event definition file created in the previous step. We have now specified that when the event is published

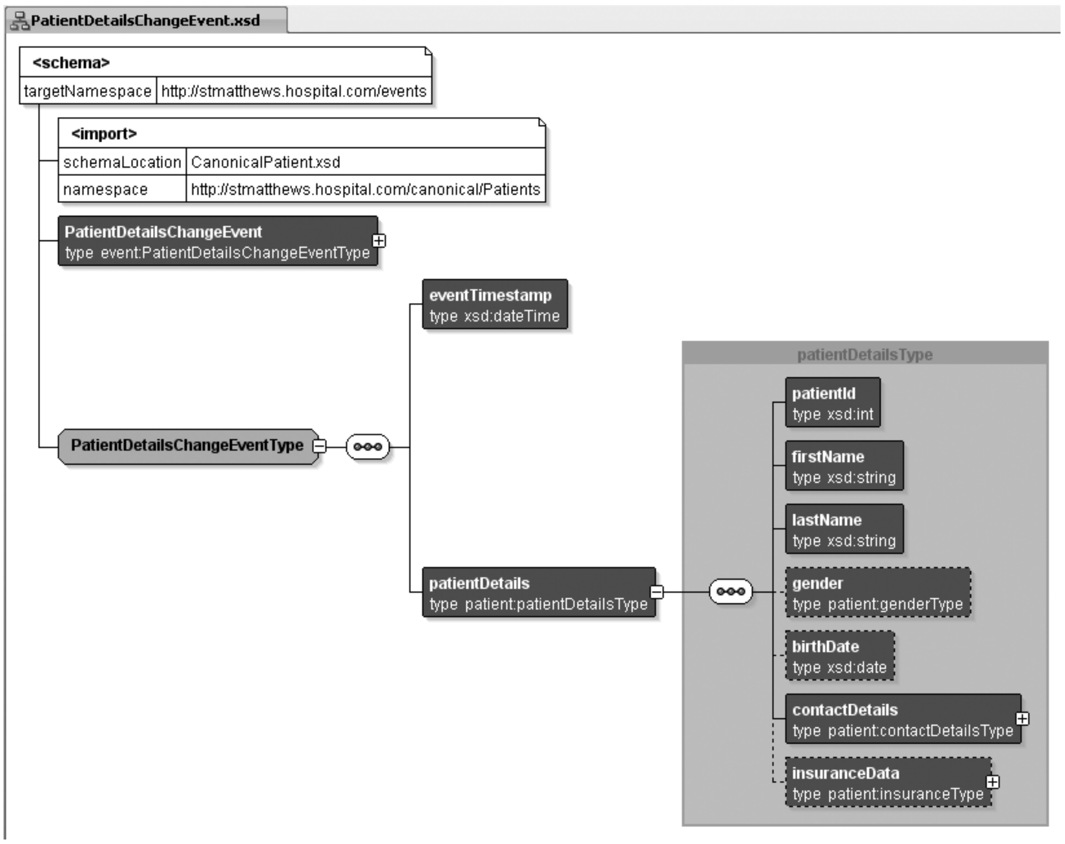
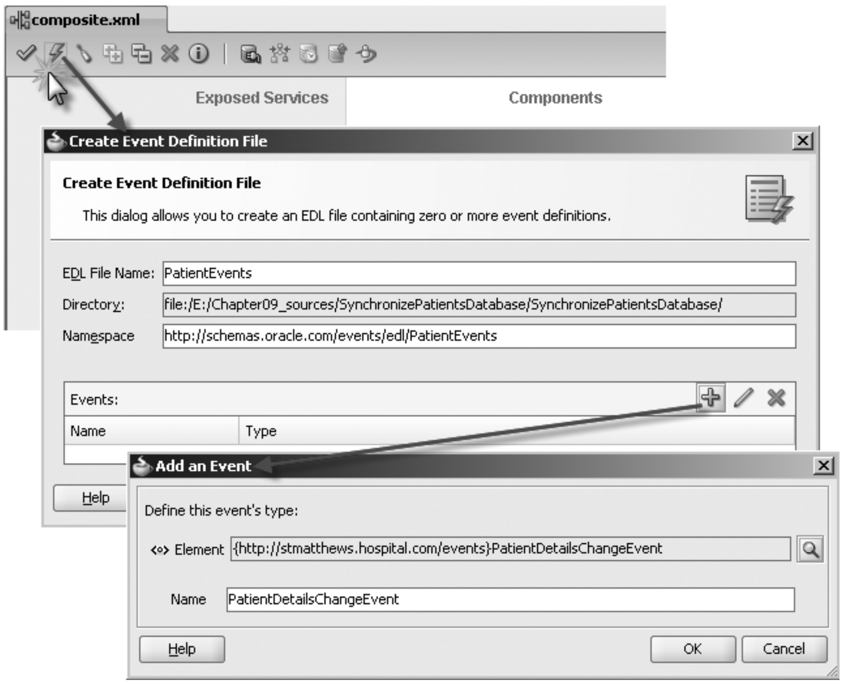

PatientDetailsChangeEvent.xsd
<schema>
targetNamespace http://stmatthews.hospital.com/events
<import>
schemaLocation CanonicalPatient.xsd
namespace http://stmatthews.hospital.com/canonical/Patients
PatientDetailsChangeEvent
type event:PatientDetailsChangeEventType
eventTimestamp
type xsd:dateTime
PatientDetailsChangeEventType
patientDetails
type patient:patientDetailsType
patientDetailsType
patientId
type xsd:int
firstName
type xsd:string
lastName
type xsd:string
gender
type patient:genderType
birthDate
type xsd:date
contactDetails
type patient:contactDetailsType
insuranceData
type patient:insuranceType

FIGURE 9-3. *The PatientDetailsChangeEvent definition in PatientDetailsChangeEvent.xsd*

on the EDN, the Mediator consumes it and can process it just like any incoming request message—except, of course, it cannot send a response. The column "Consistency" indicates whether the delivery of the event is part of the Mediator's transaction is response to the event (one and only one) or whether the delivery is in separate transaction (guaranteed). The former (and default) setting means that the event delivery is rolled back when errors occur in the Mediator (and the event can be delivered again), whereas the latter causes the event to be lost for this subscription when the Mediator transaction is rolled back. The third value—immediate—specifies that events are delivered to the subscriber in the same global transaction and same thread as the publisher, effectively (and usually very undesirably) coupling the publisher to the consumer. Stay away from that option, unless you have a very good reason for using it.

The column "Run as publisher" is set to yes to have the Mediator executed with the same security context (identity) as the composite that published the event. According to the documentation, alternatively an Enterprise Role can be set to execute the Mediator with. However, the IDE does not appropriately support this.



This dialog allows you to create an EDL file containing zero or more event definitions.

EDL File Name: PatientEvents
Directory: file:/E:/Chapter09_sources/SynchronizePatientsDatabase/SynchronizePatientsDatabase/
Namespace http://schemas.oracle.com/events/edl/PatientEvents

Events:

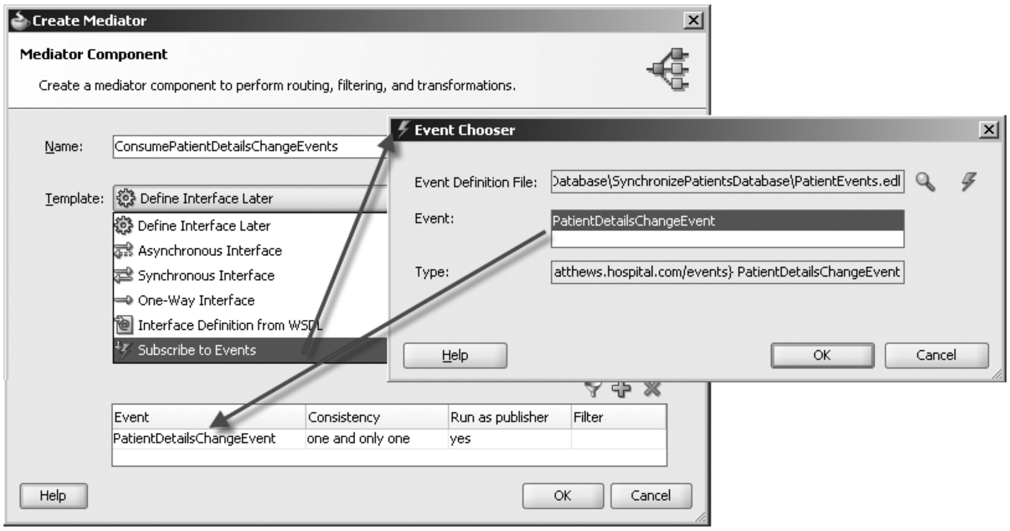
Name	Type

Define this event's type:

Element {http://stmatthews.hospital.com/events}PatientDetailsChangeEvent

Name PatientDetailsChangeEvent

FIGURE 9-4. *Opening the Create Event Definition File window and creating the PatientDetailsChangeEvent*



Create a mediator component to perform routing, filtering, and transformations.

Name: ConsumePatientDetailsChangeEvents

Template: Define Interface Later

Event Definition File: Database\SynchronizePatientsDatabase\PatientEvents.edl
Event: PatientDetailsChangeEvent
Type: atthews.hospital.com/events} PatientDetailsChangeEvent

Event	Consistency	Run as publisher	Filter
PatientDetailsChangeEvent	one and only one	yes	

FIGURE 9-5. *Creating the Mediator that consumes occurrences of PatientDetailsChangeEvent*

5. Create the database adapter service that will update the patients table. A normal best practice would be to go through a database package instead of directly coupling to the table, as was discussed and demonstrated in Chapter 5. However, to see the database adapter do straight updates for once, we will adopt that approach in this situation. The database adapter service locates records based on the patient identifier and can update various details, including surname, address, and contact data.
6. Drag a database adapter service from the Component Palette to the External References lane. Call the service **UpdatePatientsTableWithChanges**. Select the database connection to Frank's patients database. Choose the Update Only operation to perform and import Table Patients. Accept all attributes for inclusion. Click Finish.
7. Wire the Mediator to the database adapter service in the Composite Editor, as shown in Figure 9-6.
8. Double-click the Mediator to edit the routing rule from event to adapter service. Create the mapping from the event payload structure to the input format dictated by the database adapter service (see Figure 9-7).

The composite is ready for deployment. Deploy it to the SOA Suite. The EDL file with the event definition is deployed along with the composite. It is added to the dictionary of business events supported by the container. When you go to the FMW Control, you can inspect the list of all events that the Event Delivery Network knows about, as well as the subscriptions to events. First select the soa-infra node in the tree navigator. Then from the context menu, select the option Business Events (see Figure 9-8). The Business Events page opens. It shows a list of all registered business events—parsed from all deployed EDL files. The Subscriptions tab lists all subscriptions from composites on events, and indicates per subscription the precise consuming component in the composite as well as the delivery method and the filter, if there is one.

We can publish instances of the events from this page, primarily to test the subscriptions. With the PatientDetailsChangeEvent selected, click the Test button to publish a "test patient details change" event and see if the SynchronizePatientsInformation composite picks it up and processes it as expected. Note that there is no other way to test this composite because it does not expose a Web Service interface.

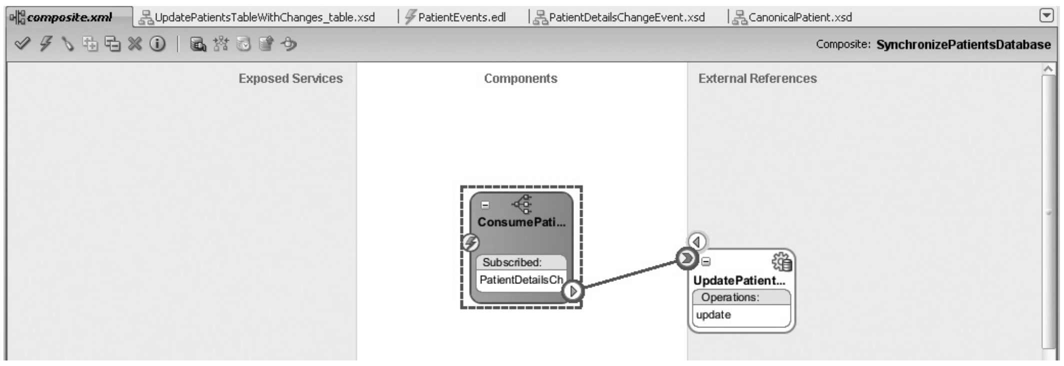


FIGURE 9-6. *Wiring the Mediator to the database adapter service*

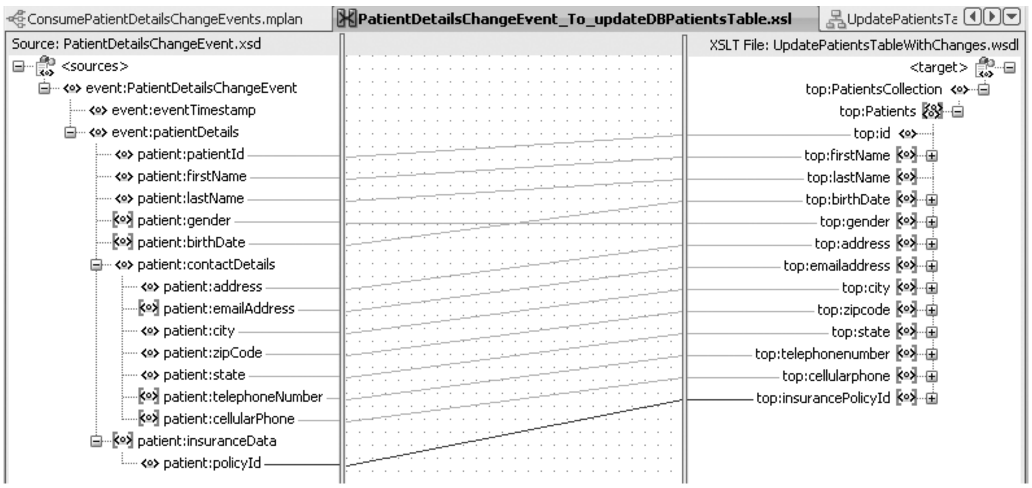
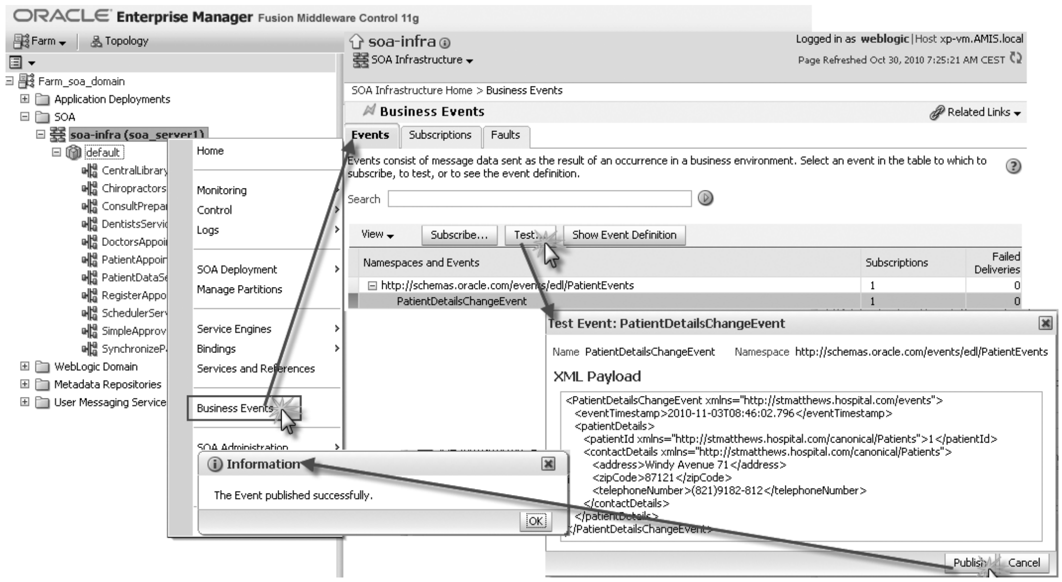


FIGURE 9-7. *Mapping the event payload to the input for the database update service*

The test instance of the PatientDetailsChangeEvent indicates that the patient with identifier 1 has moved to a new address with a new ZIP code and a new landline. He has not changed his name, undergone gender-influencing surgery, or altered other relevant details.

The publication of this event should have triggered an instance of the SynchronizePatients Information composite that has updated the patients table in the database through the database adapter service.



Events consist of message data sent as the result of an occurrence in a business environment. Select an event in the table to which to subscribe, to test, or to see the event definition.

Namespaces and Events	Subscriptions	Failed Deliveries
http://schemas.oracle.com/events/edl/PatientEvents	1	0
PatientDetailsChangeEvent	1	0

The Event published successfully.

Name PatientDetailsChangeEvent Namespace http://schemas.oracle.com/events/edl/PatientEvents

```
<PatientDetailsChangeEvent xmlns="http://stmatthews.hospital.com/events">
  <eventTimestamp>2010-11-03T08:46:02.796</eventTimestamp>
  <patientDetails>
    <patientId xmlns="http://stmatthews.hospital.com/canonical/Patients">1</patientId>
    <contactDetails xmlns="http://stmatthews.hospital.com/canonical/Patients">
      <address>Windy Avenue 71</address>
      <zipCode>87121</zipCode>
      <telephoneNumber>(821)9182-812</telephoneNumber>
    </contactDetails>
  </patientDetails>
</PatientDetailsChangeEvent>
```

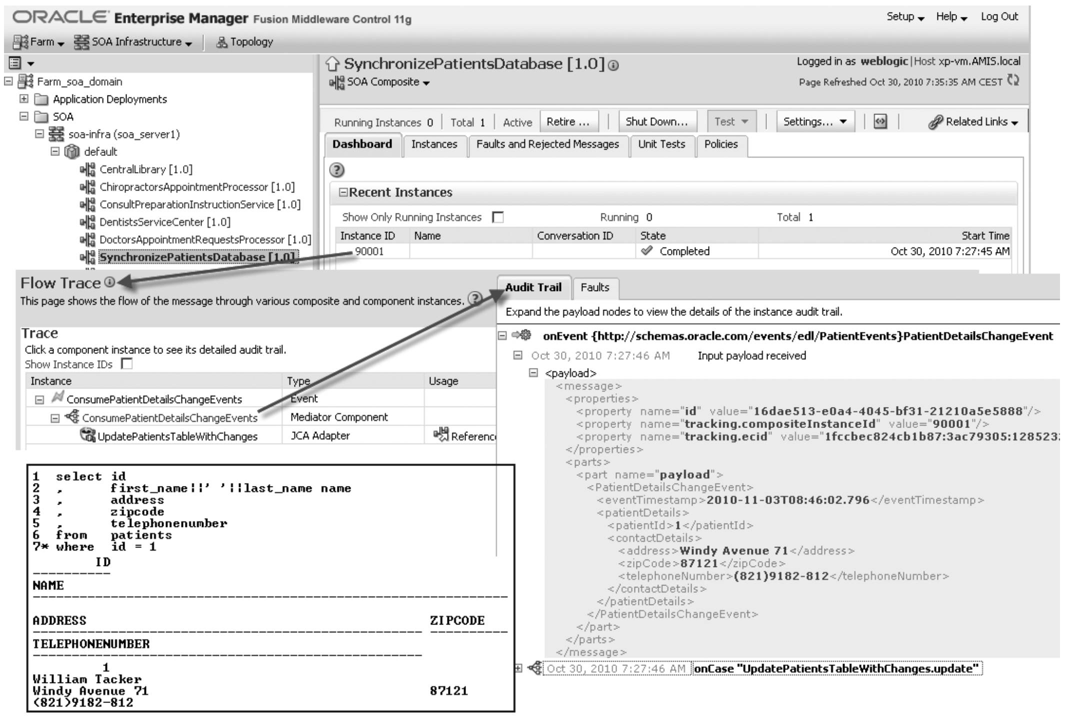
FIGURE 9-8. *Publication of a test instance of the PatientDetailsChangeEvent from the FMW control*

When we check the dashboard for the SynchronizePatientsInformation composite, we will find a new instance. Its message flow trace makes clear that it was triggered by the PatientDetails ChangeEvent—the only way it *can* be instantiated—and called the database adapter service. The message flowing into this service contains the payload from the event in a structure suited to the update service. We do not see the actual update SQL statement in the message flow. However, when we inspect the patients table contents and find the new address, we may conclude that the synchronization of the database has taken place, just as Frank envisioned (see Figure 9-9).

Frank's application does not know the origin of the event. The event appears on the EDN; the application is notified and processes the event's payload. Because the entity at St. Matthews who was first aware of the changed patient data took responsibility to publish the event, Frank's database has been synchronized—in near real time it would seem.

Other Consumers Listening In

One of the essential features of the Event Delivery Network is that the producer of the event is unaware of the consumers of the event. And, of course, that one consumer is completely independent of any other consumer. To perhaps state the obvious: Any event on the EDN can be consumed by one, multiple, or even no consumers at all. In the latter case, even if publishing the event turns out pointless in hindsight, that never relieves the producer from the responsibility to publish it anyway. When we know an event will (or even may) become important, we should publish it.


ORACLE Enterprise Manager Fusion Middleware Control 11g
Setup
Help
Log Out
Farm
SOA Infrastructure
Topology
Farm_soa_domain
Application Deployments
SOA
soa-infra (soa_server1)
default
CentralLibrary [1.0]
ChiropractorsAppointmentProcessor [1.0]
ConsultPreparationInstructionService [1.0]
DentistsServiceCenter [1.0]
DoctorsAppointmentRequestsProcessor [1.0]
SynchronizePatientsDatabase [1.0]
SOA Composite
Retire ...
Shut Down...
Test
Settings...
Related Links
Dashboard
Instances
Faults and Rejected Messages
Unit Tests
Policies
Audit Trail
Faults

SynchronizePatientsDatabase [1.0]

Logged in as **weblogic** | Host xp-vm.AMIS.local
Page Refreshed Oct 30, 2010 7:35:35 AM CEST

Running Instances 0 | Total 1 | Active

Recent Instances

Show Only Running Instances ☐ Running 0 Total 1

Instance ID	Name	Conversation ID	State	Start Time
90001			✓ Completed	Oct 30, 2010 7:27:45 AM

Flow Trace

This page shows the flow of the message through various composite and component instances.

Trace

Click a component instance to see its detailed audit trail.

Show Instance IDs ☐

Instance	Type	Usage
ConsumePatientDetailsChangeEvents	Event	
ConsumePatientDetailsChangeEvents	Mediator Component	
UpdatePatientsTableWithChanges	JCA Adapter	Referenc

Expand the payload nodes to view the details of the instance audit trail.

```
onEvent {http://schemas.oracle.com/events/edl/PatientEvents}PatientDetailsChangeEvent
  Oct 30, 2010 7:27:46 AM      Input payload received
    <payload>
      <message>
        <properties>
          <property name="id" value="16dae513-e0a4-4045-bf31-21210a5e5888"/>
          <property name="tracking.compositeInstanceId" value="90001"/>
          <property name="tracking.ecid" value="1fccbec824cb1b87:3ac79305:128523
        </properties>
        <parts>
          <part name="payload">
            <PatientDetailsChangeEvent>
              <eventTimestamp>2010-11-03T08:46:02.796</eventTimestamp>
              <patientDetails>
                <patientId>1</patientId>
                <contactDetails>
                  <address>Windy Avenue 71</address>
                  <zipCode>87121</zipCode>
                  <telephoneNumber>(821)9182-812</telephoneNumber>
                </contactDetails>
              </patientDetails>
            </PatientDetailsChangeEvent>
          </part>
        </parts>
      </message>
  Oct 30, 2010 7:27:46 AM onCase "UpdatePatientsTableWithChanges.update"
```

```
select id
,      first_name||' '||last_name name
,      address
,      zipcode
,      telephonenumber
from   patients
where  id = 1
        ID
----------
NAME
--------------------------------------------------------------

ADDRESS                                              ZIPCODE
---------------------------------------------------- ----------
TELEPHONENUMBER
----------------------------------------------------
         1
William Tacker
Windy Avenue 71                                      87121
(821)9182-812
```

FIGURE 9-9. *The instance of the SynchronizePatientsInformation composite that has updated the patients table in response to the event*

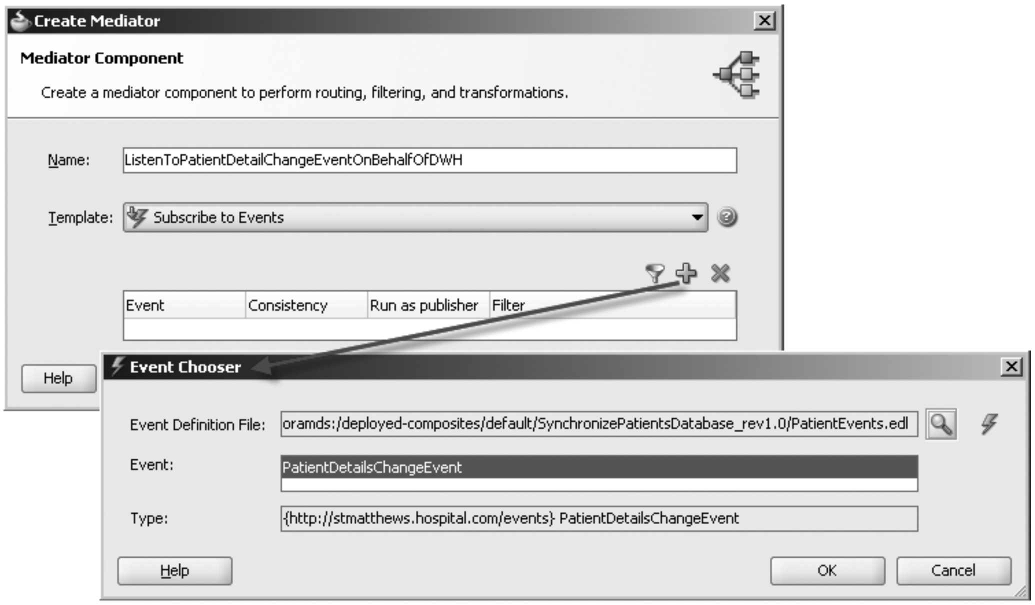
We could create a new composite application called FeedPatientChangesToDWH that consumes the PatientDetailsChangeEvent on behalf of a data warehouse that the hospital has set up, for example, to investigate demographic trends among the patient population, including moving to richer or poorer neighborhoods, marrying and divorcing, and undergoing other changes. This composite would have a Mediator that also consumes the event and propagates it to the data warehouse—through an adapter service that could talk database, JMS, AQ, or file-speak.

When we create the Mediator and indicate the Subscribe To Event template, we have to browse for the PatientDetailsChangeEvent. The most "pure" way of doing so is through the SOA-MDS connection in the Resource Palette. Through this connection, you can browse the resources in any of the composites currently deployed on the SOA Suite. Figure 9-10 shows how to subscribe the Mediator ListenToPatientDetailChangeEventOnBehalfOfDWH to the event by browsing for it in the EDL file that is deployed in the SynchronizePatientsInformation composite.

Creating Picky Subscriptions Using Filter Expressions

Even though a composite registers its interest in a certain event through an EDN subscription, this does not necessarily mean it will have to process every occurrence of the event. It is possible to define a filter expression as part of the subscription (just like we do for content-based routing in Mediators). This filter consists of a Boolean XPath expression that evaluates the event's payload, resulting in "true" for events that should be delivered to and processed by the composite and "false" for events that the composite chooses to decline.

To see the filter mechanism in action, we will help Frank decline a specific subset of occurrences of the patient details change event. It may sound unlikely, but Frank is not interested in a particular category of patients—they are simply outside the scope of his database. All patients with an identifier


Create Mediator
Mediator Component
Create a mediator component to perform routing, filtering, and transformations.
Help
Event Chooser
Help
OK
Cancel

Name: ListenToPatientDetailChangeEventOnBehalfOfDWH

Template: Subscribe to Events

Event	Consistency	Run as publisher	Filter

Event Definition File: oramds:/deployed-composites/default/SynchronizePatientsDatabase_rev1.0/PatientEvents.edl

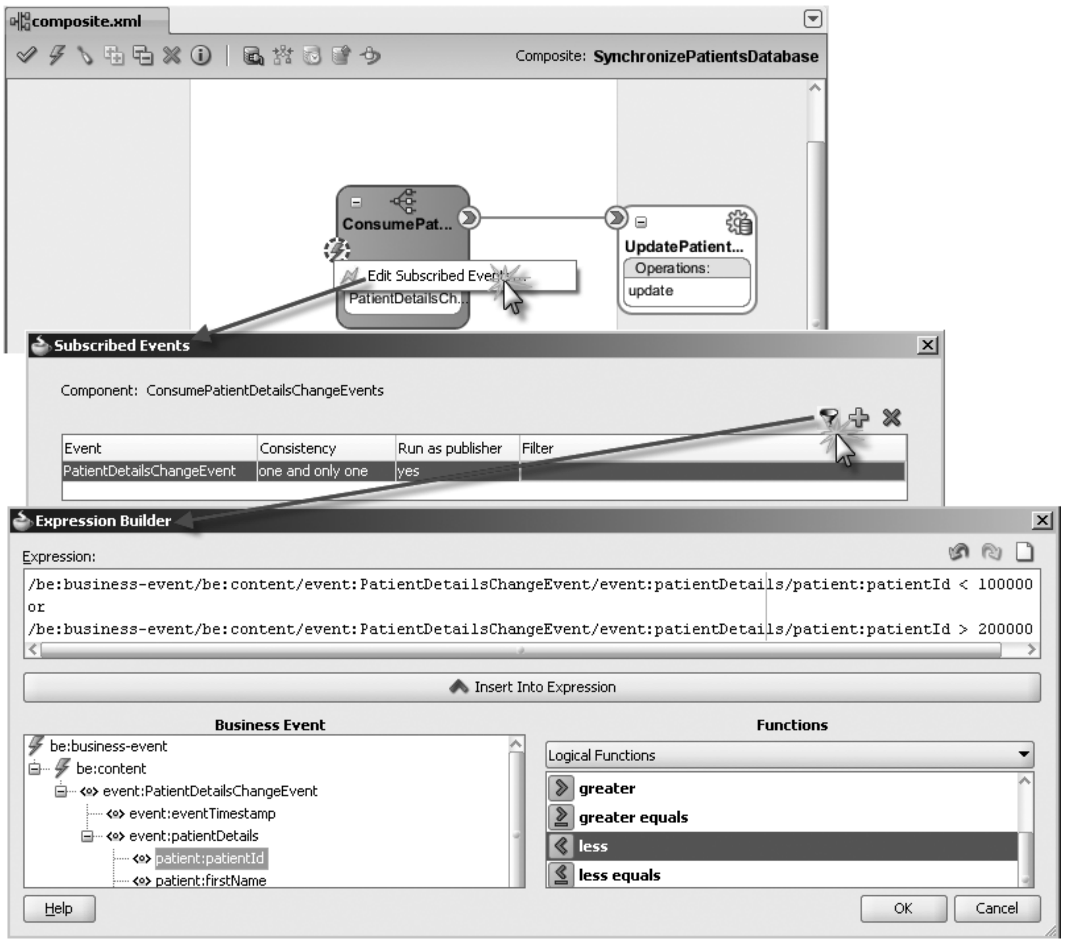
Event: PatientDetailsChangeEvent

Type: {http://stmatthews.hospital.com/events} PatientDetailsChangeEvent

FIGURE 9-10. *Browsing for an event to consume in the SOA Suite's deployed composites using the SOA-MDS connection*

between 100,000 and 200,000 are not his cup of tea (they are managed in the patient administration in the former Ophthalmology Clinic that was merged into St. Matthews several years ago). Rather than have the SynchronizePatientsInformation composite attempt updates that will have zero effect (or even result in errors) because the target record is not available, it is better to stop the event's consumption altogether. We will specify the filter that will only allow patient details change events in the meaningful range.

To begin, open the composite and then open the context menu for the event subscription icon on the Mediator. This brings up the list of event subscriptions for the Mediator, which contains only one subscription. Select the subscription and click the filter icon. This brings up the Expression Builder that helps with the creation of the XPath expression for the filter. It shows the payload structure for the event, making it easy to include the elements we need in our filter expression (see Figure 9-11).



Component: ConsumePatientDetailsChangeEvents

Event	Consistency	Run as publisher	Filter
PatientDetailsChangeEvent	one and only one	yes	

```
/be:business-event/be:content/event:PatientDetailsChangeEvent/event:patientDetails/patient:patientId < 100000
or
/be:business-event/be:content/event:PatientDetailsChangeEvent/event:patientDetails/patient:patientId > 200000
```

FIGURE 9-11. *Specifying a filter expression for the PatientDetailsChangeEvent subscription*

Create the following filter expression:

```
/be:business-event/be:content/event:PatientDetailsChangeEvent...
                        .../event:patientDetails/patient:patientId < 100000
or
/be:business-event/be:content/event:PatientDetailsChangeEvent...
                        .../event:patientDetails/patient:patientId > 200000
```

Basic Queuing vs. Business Queuing, or JMS and AQ vs. EDN

Publishing messages that are forwarded by a framework to registered consumers in order to achieve a highly decoupled exchange: That, in short, is what EDN adds to the SOA Suite. But that could also serve as a brief definition of both JMS (Java Message Service) and AQ (Oracle Advanced Queuing). And because we already had those two at our disposal without the SOA Suite, as well as technology adapters for both JMS and AQ to make them available from within the SOA Suite, one might wonder what exactly is the added value of the Event Delivery Network.

The fundamental concepts of EDN on the one hand and JMS and AQ on the other are pretty much the same. Most of the things we do with the Event Delivery Network and the supporting facilities in the SOA Suite can also be done with plain JMS or AQ—although it would require a lot more work.

JMS and AQ are not specially geared toward XML payloads or integration into the world of SOA, SCA, and Web Services. In order to work with vanilla JMS or AQ, we would have to implement technology adapter services that are listening to queues, one for every consumer of an event arriving on that queue. These adapter services would work with Mediators that play a similar role as they would when dealing with EDN events.

Before we can get started with our JMS or AQ operations, the underlying JMS queues or topics or the AQ queues would have to be created and configured in either WebLogic Server or the database.

The SOA Suite provides additional value with the EDN style of events. Event definitions are consolidated across the SOA container, defined in EDL files and available from MDS. The FMW Control presents the Business Events in a separate page that shows the events and their definitions, all subscriptions and faulted deliveries, and allows us to publish test events onto the Event Delivery Network. Events have become first-class citizens in the SOA Suite thanks to the EDN.

Many of the things we could do using JMS or AQ and would have to configure ourselves have been prebaked into the SOA Suite EDN infrastructure, making life a lot easier. The EDN allows us to focus on the contents of the events and the logic of event handling, abstracting away the finer technical details of JMS or AQ. However, the Event Delivery Network internally has been implemented on top of these standard messaging technologies. EDN has two different implementations—namely, EDN-DB and EDN-JMS. EDN-DB uses an Oracle Database as a back-end store and depends on Oracle-specific features. EDN-JMS uses a vanilla JMS queue as a back-end store. By default, when the dehydration store runs on an Oracle database, you get the AQ-based EDN-DB, and otherwise the JMS one.

Close the Expression Builder and the Subscribed Events window. Now the composite can be redeployed and we can publish test events to see whether the filter blocks out the events regarding patients in the 100,000–200,000 range.

If we were to deploy the composite as a new revision, we would see something that might seem odd at first. PatientDetailsChangeEvents for identifiers between 100,000 and 200,000 are still consumed and processed, despite the filter expression that should block them out. Events with identifiers outside that range are processed twice by the SynchronizePatientsInformation composite—or rather by the two revisions of the composite.

Events are picked up by all revisions of a composite application—not just the last or the one marked as default! (See Chapter 17 on versions of composite applications.) To receive the event with the latest revision of the composite only, it is recommended that you retire all previous revisions of the composite.

Publishing Patient Details Change Events

Mediators can easily be configured to consume events, as we have seen in the previous section. But they are just as capable of publishing events. A routing rule can be created that forwards a message in the form of an event's payload to the Event Delivery Network, instead of to a target service. Alternatively, you can open the context menu on a Mediator in the Composite Editor and open the Event Chooser by clicking the option Add Published Events (see Figure 9-12). The routing rules are created automatically.

A component such as a Mediator does not need a registration or configuration with the EDN before being able to publish events to the EDN. The Mediator can just send one to the EDN. You will not find a list of event publishers in the FMW Control.

Frank not only consumes patient details change events, he also publishes them to the EDN in situations where updates to patient records first appear in his database. Of course, we need to make sure that no events are published from an update to the database when that update is indirectly the result of another patient change event on the EDN—or that Frank consumes his own events, thus ending up in an infinite loop.

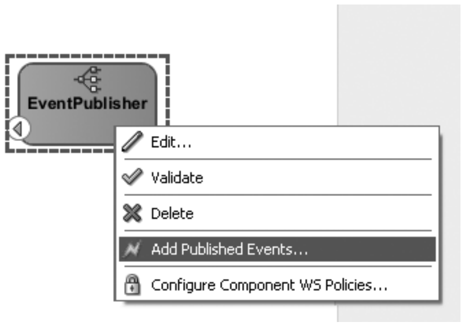


FIGURE 9-12. *Adding published events to a Mediator*

Publishing Database Events via Advanced Queuing

Ultimately, all changes to Frank's patient database end up in the database itself. All data changes can therefore be captured at the core level using a database trigger against the patients table. We would then have to somehow upgrade that database-level DML (data manipulation) event to an event that the EDN can handle. A good, decoupled approach would be to use Advanced Queuing (AQ) to publish the DML event from a database trigger—as shown in Figure 9-13. The database application has the responsibility for publishing the event, but it does not get coupled to the specific ways of the SOA Suite and its proprietary EDN requirements. The trigger uses a concept from its own realm—the Advanced Queue—and what happens next with the published event is out of its hands and mind. However, anyone listening in on the Advanced Queue can process the event in any appropriate way.

From the SOA Suite end, we would use the AQ Technology Adapter to listen to these events that get published on the queue (or multiconsumer topic). This adapter service would forward the event's payload to a Mediator that will then publish the event as described in the EDL on the Event Delivery Network.

Alternatively, the database trigger could also make use of the EDN's PL/SQL API to publish a real EDN event straightaway. That is probably easier than doing the AQ thing. However, it couples the database application to the SOA Suite in a way that could be seen as too restrictive: The availability of the SOA Suite would become a requirement to run and even compile the PL/SQL code.

Java applications can adopt a similar setup with JMS as the pub/sub queuing mechanism instead of Advanced Queuing. In this case, the JMS technology adapter consumes messages from the JMS queue and propagates them to a Mediator that publishes to the EDN. Java applications could alternatively use the EDN's SendEvent Java API.

Another approach is one where database applications and Java applications both could call a very generic, one-way Web Service exposed by a Mediator that turns various request messages into different types of EDN events.

The wiki contains a complete example of a database trigger that feeds DML events into an Advanced Queue and an AQ adapter service listening to these events and forwarding them to a Mediator that publishes events on the EDN. Another example demonstrates the same approach from a Java application using JMS and the JMS technology adapter.

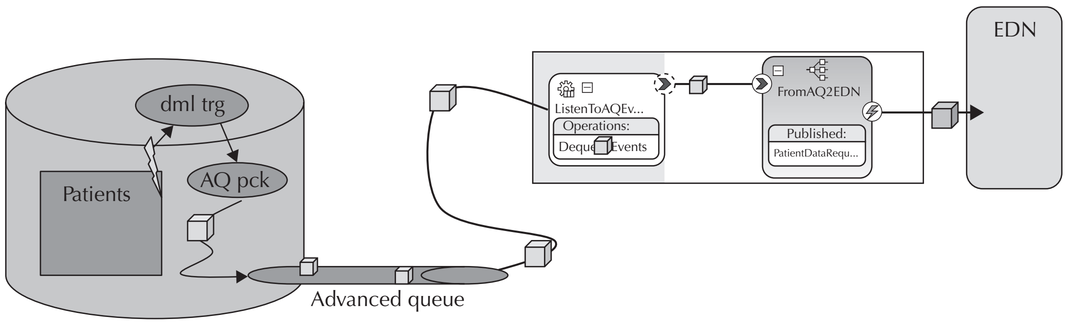


FIGURE 9-13. *Publishing events from the database to the EDN via Advanced Queuing*

Advanced Queuing and JMS are, of course, perfect ways to share events that appear on the Event Delivery Network with consumers outside the SOA fabric—either Java (JMS) consumers or PL/SQL (AQ) consumers. In these cases, a Mediator consumes the EDN event and forwards it to a technology adapter for either JMS or AQ that puts it on the respective queue or topic.

Chapter 20 demonstrates ADF Business Components that can publish EDN events, either in the same or a different container from the one running the SOA Suite.

Publishing EDN Events from BPEL Components

The PatientDataService composite developed in Chapter 5 only supported operations to retrieve a patient record. That application has been extended and now exposes a service for creating a new patient record. This service can be invoked, for example, from the PatientAppointmentService composite that sometimes receives appointment requests for unknown patients, for whom a new record should be created (see Figure 9-14). The operation to create a new record expects a request message that contains all data for the new patient. It will respond with a message that contains the newly assigned patient identifier. Internally it uses a new database adapter service that inserts a record into the patients table and the existing service that retrieves the patient's ID based on the combination of first name and last name.

After the BPEL component ProcessNewPatient has called the database adapter services, first to create the new patient record and then to retrieve the patient identifier that was assigned during the insert from a database sequence, it would be a good moment to publish a PatientDetails ChangeEvent to inform potential consumers of this new patient record. Publishing an event from a BPEL process is a bit special, because BPEL is an open standard and the EDN is not. Oracle has extended BPEL—which is supported by the standard—to support additional activities such as

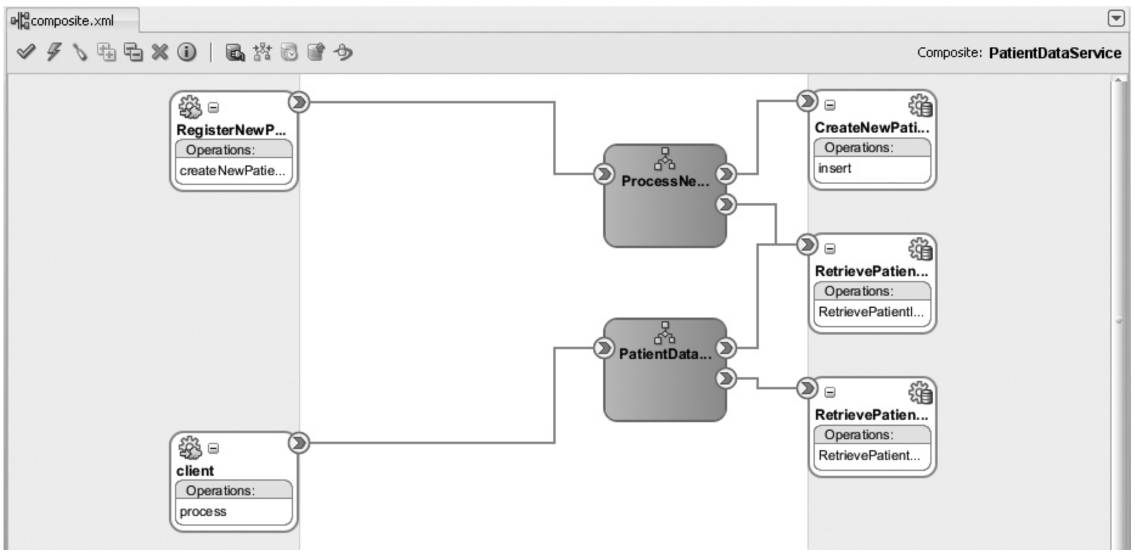


FIGURE 9-14. *PatientDataService composite with support for the "create new patient" operation*

special versions of Receive and onMessage for consuming events as well as special versions of Invoke to publish an event. Also see the next section for some details on these extensions and the evolution of the SCA standard with regard to events. In general, it seems best to use a mediator to either receive or publish events, unless those events are required for correlation purposes: When an event must be consumed by a specific running instance of a composite application, the event consumer should be the BPEL process component.

If we want the BPEL process to publish an event, we have to add an Invoke activity, dropping it after the scope RetrieveNewlyAssignedIdentifier (see Figure 9-15). Call the activity Publish_PatientDetailsChangeEvent. Select the appropriate interaction type of event. The editor changes and now allows us to browse for the event that we want to publish—using the Event Chooser dialog we have used before. In this case, select the PatientDetailsChangeEvent from the PatientEvents.edl found in the SOA-MDS connection in the deployed composite SynchronizePatients Information. After selecting the event, click the green plus icon to have a local variable created that we will set up with the event's payload.

We need to have the XSD document available in our project that contains the definition of {http://stmatthews.hospital.com/events}PatientDetailsChangeEvent. Otherwise, JDeveloper's editors will not know the structure of the variable. We can copy that file (or refer to it), either from MDS or from the SynchronizePatientsInformation application on the file system, to the xsd directory. After creating the BPEL process using the design-time editors, we could remove that file again.

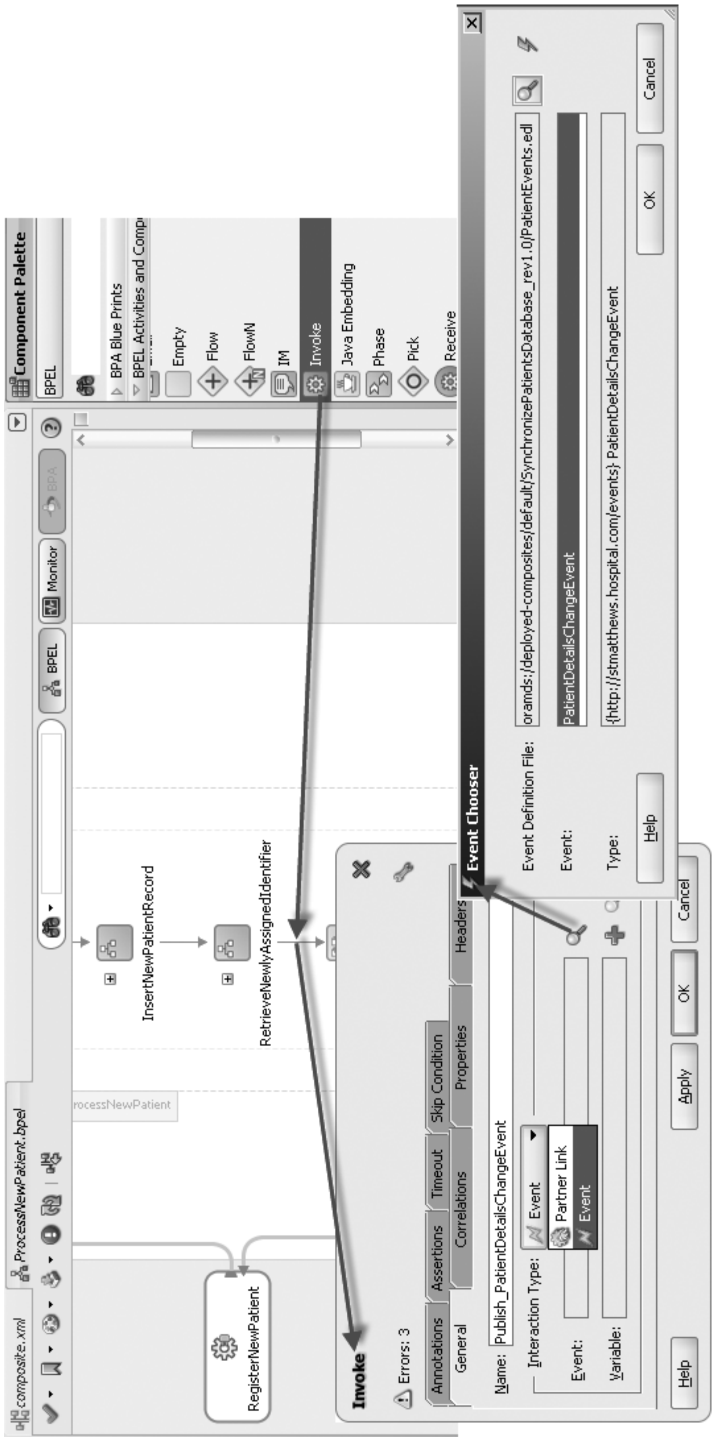
We will now add an activity that will initialize the variable PatientDetailChangeEvent_payloadVariable. We could use an Assign activity with a dozen Copy steps, but it's preferable to use the Transform activity to map the contents from the inputVariable to the variable for the event payload. We then need an additional Assign activity with a single Copy operation to set the patientIdentifier.

With the transform, assign, and invoke (or publish) added to the process, we are ready to deploy the revamped PatientDataService—and find out whether it will publish events to the Event Delivery Network.

Testing the publication of events to the EDN is really only possible when there are consumers of the event. There is no way to track publication of individual events, either through the FMW Control or in any other way, except for the audit trail on the end of the consumers of the event. On the publishing side, there is only an audit trail for events that have faulted delivery.

In this case, we have a consumer of the event—or even two—so we can test whether the BPEL process publishes events as it is supposed to (see Figure 9-16). When we invoke the PatientDataService to have it create a new patient, after making the database adapter perform the insert of the new record, the BPEL process publishes the PatientDetailsChangeEvent that subsequently gets consumed by two composites: FeedPatientChangesToDWH and SynchronizePatientsInformation. The former writes an entry in the log file, and the latter performs an update of the database record that was just created—with the values it already has. Pointless but harmless. In this case, we should add an indication to the event about its origin as well as a filter to the event subscription in the SynchronizePatientsInformation composite to not consume instances that were published by Frank's own Patient Data Service.

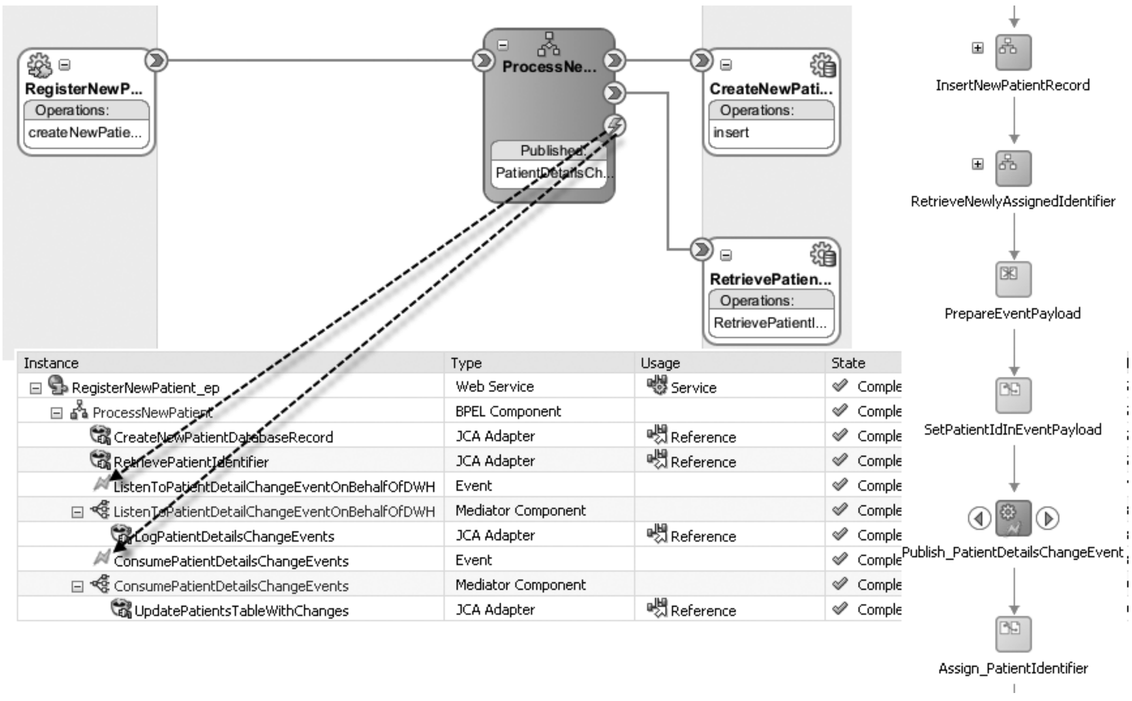
Note how the FMW Control message flow trace includes both the composite instance that published the event as well as the two instances for the consuming composites—even though there is no direct relationship between these three composites.



Name: Publish_PatientDetailsChangeEvent
Interaction Type: Event
Event:
Variable:

Event Definition File: oramds:/deployed-composites/default/SynchronizePatientsDatabase_rev1.0/PatientEvents.edl
Event: PatientDetailsChangeEvent
Type: {http://stmatthews.hospital.com/events} PatientDetailsChangeEvent

FIGURE 9-15. *Adding the Invoke activity to the BPEL process that will publish the event*



Instance	Type	Usage	State
⊟ RegisterNewPatient_ep	Web Service	Service	✓ Comple
⊟ ProcessNewPatient	BPEL Component		✓ Comple
CreateNewPatientDatabaseRecord	JCA Adapter	Reference	✓ Comple
RetrievePatientIdentifier	JCA Adapter	Reference	✓ Comple
ListenToPatientDetailChangeEventOnBehalfOfDWH	Event		✓ Comple
⊟ ListenToPatientDetailChangeEventOnBehalfOfDWH	Mediator Component		✓ Comple
LogPatientDetailsChangeEvents	JCA Adapter	Reference	✓ Comple
ConsumePatientDetailsChangeEvents	Event		✓ Comple
⊟ ConsumePatientDetailsChangeEvents	Mediator Component		✓ Comple
UpdatePatientsTableWithChanges	JCA Adapter	Reference	✓ Comple

FIGURE 9-16. *Testing the creation of a new patient record and subsequent publication of Patient DetailsChangeEvent by the BPEL process*

Complex Event Processing

The SOA Suite license encompasses a product called Complex Event Processing (CEP); however, this is not integrated into the SOA Suite run time (it runs in its own stand-alone, lightweight, specialized container). CEP is not a service engine, and we cannot embed CEP service components in our composite applications. CEP clearly deals with events—so how does it relate to the Event Delivery Network?

Complex Event Processing—discussed in more detail in Chapter 19—deals with high volumes of often very small events (in terms of payload), which are possibly by themselves meaningless. Events processed by CEP can be as frequent and small as RFID sensor detections, stock ticker changes, hits on a page on a website, temperature or humidity readings from a physical sensor, DML events in the database, luggage passing detectors on the airport's conveyor belts, or products being scanned on the cash register. CEP tries to find patterns, derive aggregates, and detect deviations in the streams of events that it processes. The conclusions it may arrive at are usually forwarded as a real business event to a specific consumer or to a queue that consumers may register to. To be sure, the "Complex" in CEP refers to the nature of the processing (algorithms and logic), not to the events themselves, because those are usually extremely simple!

Consider a volleyball match as an example of the difference between the simple events CEP processes and the derived or promoted business events. In this match, every point scored amounts to a simple event (or even every individual ball contact, when gathering match statistics), and only the conclusion of a set or even of the entire match warrants a business event.

The events travelling on the Event Delivery Network usually are business events—with more meaning and larger payloads than the typical event processed by CEP. Events on EDN occur with typically much lower frequencies, orders of magnitude below those of the CEP event streams. However, the results produced from Complex Event Processing may very well be promoted to or published as events on the EDN, propagated via a JMS queue or the EDN's Java API to the SOA Suite.

Event Delivery Network in SCA and BPEL

The definition of events and the publication of and subscription to these events are recorded in a number of files that are read, parsed, and interpreted by the SOA Suite and more specifically by the Event Delivery Network.

The events are defined in one or more EDL (Event Definition Language) files, which are deployed inside composites or stored in MDS.

The subscription to an event, visualized in the Composite Editor through the little lightning icon, is recorded in the composite.xml file. In this file, components can have the child element "business-events." This element, in turn, can contain one or more occurrences of the "publishes event" element as well as one or more instances of the subscribe element.

Analyzing the SCA Configuration Around EDN and Events

The entry in the composite.xml file for the BPEL component ProcessNewPatient is as follows:

```
<component name="ProcessNewPatient">
  <implementation.bpel src="ProcessNewPatient.bpel"/>
  <business-events>
    <publishes xmlns:pub1="http://schemas.oracle.com/events/edl/PatientEvents"
               name="pub1:PatientDetailsChangeEvent"/>
  </business-events>
</component>
```

It is clear to see how this component is configured to publish instances of the PatientDetails ChangeEvent. In a similar way, we find from the entry of the Mediator ConsumePatientDetails ChangeEvents in the composite.xml file for the SynchronizePatientsInformation application that it subscribes to the PatientDetailsChangeEvent, but it filters on a specific condition.

The composite.xml contains the high-level associations between the composite and the events that get consumed or published. The details that specify exactly when and with which payload an event is published are not defined at that level, but rather at either the Mediator or the BPEL process level.

The SOA Suite run time knows how to interpret the event definition in the EDL document. It also parses the event subscriptions from the deployed composite applications, both Mediator and BPEL components, so it knows where to send events of specific types when they occur. The next paragraphs describe the extensions to BPEL and Mediator configurations with regard to events.

BPEL Extensions for Consuming and Publishing Events

Oracle has extended BPEL—in a way that is prescribed in the BPEL standard specification—to implement a special type of Invoke activity that does not call out to a partner link, but instead publishes an event to the EDN.

The code in the BPEL process ProcessNewPatient that publishes the event is extremely simple—an Invoke element with a special attribute called eventName in the http://schemas.oracle.com/bpel/extension namespace that is interpreted by the JDeveloper design-time tools as well as the SOA Suite run-time engine for BPEL to refer to an event defined in one of the EDL files, rather than a partnerLink defined in a WSDL:

```
<invoke name="Publish_PatientDetailsChangeEvent"
bpelx:eventName="ns6:PatientDetailsChangeEvent"
        inputVariable="PatientDetailChangeEvent_payloadVariable"/>
```

A similar extension is used in the Receive and onMessage activities when they are to consume an event instead of an incoming request message.

Consuming and Publishing Events in Mediator mplan Configuration

The Mediator configuration file (mplan) has a proprietary format—there is no industry standard underpinning the mediator's operations (or the Mediator concept itself in relation to SCA). The mplan file has an eventHandler element for Mediators that subscribe to an event (instead of a request message):

```
<Mediator name="ListenToPatientDataRequestEvent"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.oracle.com/sca/1.0/mediator">
  <eventHandler xmlns:sub1="http://schemas.oracle.com/events/edl/PatientDataEvents"
        event="sub1:PatientDataRequestEvent"
        deliveryPolicy="AllOrNothing" priority="4">
```

It uses a raise element in routing rules that publish an event to the EDN instead of forwarding a message to a target service:

```
<raise xmlns:pub1="http://schemas.oracle.com/events/edl/PatientDataEvents"
       event="pub1:PatientDataResponseEvent"/>
```

Events and Publish/Subscribe in the SCA Specifications

The events and the publish/subscription model we have just seen in action in the Oracle SOA Suite are not part of the original SCA specifications. Only fairly recently did discussions take place in the OASIS SCA committee regarding events, partly instigated by Oracle's representatives. This has led to a specification—published in April 2009—that the OSOA community considers final. For more information on this specification, see the link on the book's wiki to the SCA Assembly Model Specification Extensions for Event Processing and Pub/Sub.

This extension to the SCA specifications details the description of consumers and producers of events in the composite.xml file, very much like the business-events element Oracle currently uses in the composite.xml document for specifying which events are consumed and published by a component.

The definition of events in SCA is similar to the EDL format used in the Oracle SOA Suite. Events have a qualified name and a reference to an XSD element that describes the shape of the event data. The specification also speaks about optional additional metadata associated with event types, such as creation time. The SOA Suite documentation mentions custom headers—which seem similar beasts to these optional metadata elements. However, the custom headers do not seem to currently have a working implementation. The SCA definition also defines filters as a way for consumers to fine-tune their interest in events. A filter inspects the event type, event metadata, and event business data (the payload) and uses expressions—which could be XPath, although other languages are allowed in the specification—that evaluate to true when an event should be accepted by the consumer.

The main distinction between the SOA Suite Event Delivery Network and this SCA extension specification is the concept of a channel in the SCA specification. A *channel* is an intermediary between producers and consumers of events. Channels can be used inside composite applications or at the domain level. Channels can be used for a subset of the full set of event types flowing through the SCA container. Note that filters can be applied to channels as well as to consumers. Channels are primarily a means of organizing and administering the event infrastructure of the SCA container, without adding business functionality. The EDN itself is similar to the *default channel,* and currently the SOA Suite does not support alternative channels.

It seems likely that the Oracle Event Delivery Network will morph into an implementation of this SCA extension for events. That will mean only a slight change in the metadata files (composite.xml and the EDL files) because the logic of events, publication, and subscription remains the same.

Alternative Ways for Publishing Events to the EDN

The most obvious way to publish an event to the EDN is from a Mediator or BPEL service component. For testing purposes, the FMW Control also provides a "publish test event" facility. You can also publish an event from an Ant target, like this:

```
<target name="publishPatientDataRequest">
   <java classname="oracle.integration.platform.blocks.event.SendEvent"
         fork="true" failonerror="true">
      <classpath>
         <pathelement path="${soaEDN.classpath}"/>
      </classpath>
      <arg line="-dbconn localhost:1521:orcl
                 -dbuser FMW_SOAINFRA
                 -dbpass oracle
                 -event patientDataRequestEvent.xml"/>
   </java>
</target>
```

Details for this call, including the setting for the soaEDN.classpath property and the contents of the patientDataRequestEvent.xml file, can be found on this book's wiki.

The SendEvent class used from Ant can also be leveraged programmatically from a Java application to publish an event to the Event Delivery Network. An example is shown on the wiki.

Chapter 20 on ADF explains how ADF Business Components can be configured in a declarative way to publish events associated with data manipulations on Entity objects. Finally, there is a PL/SQL procedure—in the FMW SOA Infrastructure schema—called edn_publish_event with four input parameters: local_name (of the event; for example, PatientDataRequestEvent), namespace (in which the event is defined, such as http://schemas.oracle.com/events/edl/PatientDataEvents), payload (the XML fragment that should go inside the <content> element), and priority (an integer that does not yet seem to have any effect).

Of course, there are many indirect ways to publish events: A Mediator can publish EDN events—and it can be triggered by database, file system, FTP, AQ, and JMS adapters alike or be invoked through the SOA Suite Java API, SOAP, or REST binding.

Decoupling Two-way Services Using the Discussion Forum Approach

In the previous pages, we have used the Event Delivery Network to decouple producers of business events from all interested consumers. The only responsibility we assign to applications that know about or even create new information is to have them publish business events to the central Event Delivery Network of the SOA Suite. These events should have a payload with enough information to make them meaningful to potential consumers. However, the publisher does not concern itself with who might be interested in the event and what those interested parties will do with it. The publisher never has to look back after publishing the event—and certainly not wait for a response! What's more, we can add or remove publishers and consumers without any impact on them.

That, by and large, resolves the decoupling challenge for the one-way services. Then there is still the situation where composite applications call two-way Web Services, because they need a response from those services. This situation introduces some coupling: The calling application depends on the service contract—the message types, portType, and operation names—and needs to know where to find the service. In this case, because the calling application has its own reasons for making the call, this moderate level of coupling is usually acceptable. However, the Event Delivery Network makes it possible to eliminate these direct dependencies in many situations by using the Discussion Forum pattern.

Introducing the Discussion Forum Pattern

To completely decouple service providers from service consumers, we have to rethink the concept of soliciting a response from the service provider. Instead of making a direct call to a specific service to have a specific question answered, we can publish an event that contains the service request to the EDN like a generally broadcasted cry for help. The event publisher does not know who will be able to handle the request and formulate the response. It does not know where the service is located that is capable of handling the request, what its contract is, and whether it is synchronous or asynchronous, if it is available (and if not, whether there is a fallback alternative). It assumes—and that is probably the big challenge with this approach—that some entity consumes the event, processes the request that is contained in it, and publishes the response in the form of a new event.

The original request event contains a special question identifier that is used in the response event and allows the requesting application to pick up the response to its original question.

This approach seems a bit similar to one of the many discussion forums on the Internet—for example, the OTN Discussion Forums. Such forums can produce a response to any question you may ask, sometimes extremely rapidly. And most forums have a mechanism that will send an e-mail to the original requester when a response has been provided. However, sometimes a question is answered only after a fairly long period—or never at all. A forum moderator could monitor questions that do not get answered and take appropriate action. Our composite applications will probably not suffer a lack of response very well, so we may need to ensure that all question events are followed up with response events, if necessary, through an SOA Suite counterpart to the forum moderator.

A First Stab at the Decoupling from a Two-way Service Using the Discussion Forum Approach

Let's see what we need to do to implement the discussion forum approach for getting hold of patient data. At this moment, applications that want to have details on a specific patient can invoke the synchronous PatientDataService that Frank developed in Chapter 5 using BPEL and a database adapter service. We see the coupling aspects that we described earlier: Anyone interested in patient data needs to know about the existence of the PatientDataService, the details of its contract, and the physical location of its endpoint. We are now going to extend the PatientDataService application to make it participate in the more decoupled, discussion forum style of serving up patient details.

The Discussion Forums Pattern in Action

Let's assume a BPEL process that has a need for some patient data. It already knows the patient identifier, but it does not know of any specific service that can provide the patient data. However, it has heard that when one publishes a PatientDataRequestEvent to the EDN, there will be helpful souls out there that may respond to the call for help and publish an event of their own: Patient DataResponseEvent. This response event can be correlated on the eventGUID that was injected into the request message and—presto—there's the data being looked for. This pattern is illustrated by Figure 9-17.

We will work with the PatientInvoiceProcessing composite application. It is intended for creating billing statements for individual patients. This application publishes instances of PatientDataRequestEvent. The PatientDataService composite is extended with a Mediator that subscribes to this type of event and that will publish the PatientDataResponseEvent with the required information.

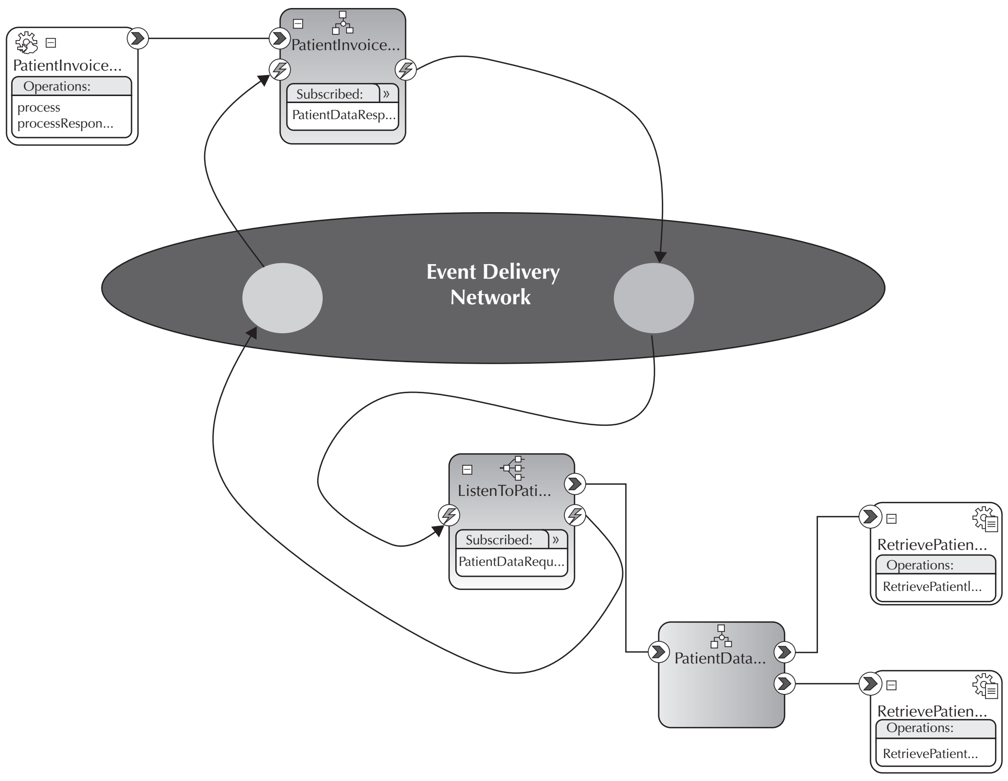


FIGURE 9-17. *The Discussion Forum pattern: soliciting a response from an unknown provider by publishing an event*

The online chapter complement contains the detailed, step-by-step implementation of this.

To then see the discussion forum in action, go to the FMW Control, locate the PatientInvoice Processing composite, and test it, providing some patientIdentifier value. The BPEL process is invoked and will publish a PatientDataRequestEvent to solicit more details on the patient.

The event is consumed by the PatientDataService composite, which processes it and publishes a PatientDataResponseEvent that contains the requested patient data as well as the eventGUID that was sent in the original request. The BPEL component is waiting to receive this response event, and will correlate it on the eventGUID to ensure that the event is consumed by the same instance that sent out the original request event. Note that other consumers could consume the same event—although it probably would not make sense to them.

Extending the Event-based Patient Data Service

We could make things even more interesting by extending the functionality we have implemented so far. We can, for example, introduce a second composite application willing to process request events—only for patient identifiers over 50,000. This composite would have a filter in the event subscription to only accept the over-50,000 identifiers. It would be interesting to see what happens when both consumers handle the request and produce a response event. The first response event would probably correlate with the requesting BPEL process, and the second one simply fades out on the EDN.

We can also add a moderator of sorts—a Mediator that consumes request events and stores them in a database table, along with the date and time. Another Mediator would then consume response events and use them to update the request event records in the database. Request records that have not yet been updated are the unanswered ones that may need moderation.

Responses to "cries for help" can be fairly large. In general, we would like to keep the size of the event payload as small as possible, as to not overburden the service fabric. Instead of sending the full response in the response event, as we have done here, we can use the claim-check pattern for the response: Do not actually send the patient data in the response message; rather, send a unique identifier (claim-tag) that can be used to retrieve the XML document with patient data at some generic, central service that hands out XML documents based on unique identifiers (a bit like the YouSendIt Internet service for handling large attachments by sending a URL from which the attachment can be downloaded rather than sending the attachment itself); the PatientDataService would have to register their XML document with this central document service before publishing the response event.

Judging the Discussion Forum Pattern

This discussion forum, like the event-based reply-response pattern, is clearly not suitable for every situation. First of all, it is an asynchronous pattern that requires the requestor to consume events from the EDN.

Second, the event carrying the response needs to be correlated with the application instance that asked the original question. In the SOA Suite today, only BPEL components can consume events and correlate them to running instances.

The decoupling achieved through this event-based mechanism comes at the price of performance and implementation overhead. A direct, synchronous call to a known service will yield a response much faster than this asynchronous approach that requires the Event Delivery Network as the intermediary for both the request and the response.

These should be serious considerations before adopting this pattern. The primary reason for discussing this pattern—which is really a corner case—is to demonstrate what level of decoupling is attainable through events.

Having Non-Events Published on the EDN

An event indicates that something happens—or so you would think. However, it is quite possible to have a meaningful event when something *does not* happen. When the expected does not occur, that can be quite an event indeed. If the sun or the tide forgot to rise, we would have major events indeed.

On a slightly smaller scale, we, too, have the situation where the fact that something we expected did not actually happen in itself is a meaningful event. The invoice was sent but the payment never materialized; the suitcase went into the baggage-handling system but never came out; the complaint was filed but a response was never sent. Such an event is often called a *non-event*. The absence of an event is an event in its own right.

Non-events are frequently associated with real events: The (real) event takes place and normally has a partner event. The suitcase entering the system is the real event, and the same piece of luggage ending up in an airplane could be its partner event. The absence of this partner event (after a set period) triggers the non-event. A person entering a secure zone is the real event that should partner with the event of that same person leaving the secure zone at some point. The non-event originates when the person does not reappear from the secure zone after some predefined period. Complex Event Processing (CEP) deals with analyzing enormous volumes of events, and one of the goals of CEP is the detection of non-events. Sensors not sending the signal they were supposed to send, the container not appearing at the next RFID sensor station, the car not exiting the tunnel. (More on CEP appears in Chapter 19.) Note that BAM (Business Activity Monitoring), also discussed in Chapter 19, can identify non-events, too.

St. Matthews tries to find the balance between providing emergency care to everyone and safe-guarding the financial budget. Everyone who needs immediate medical attention can be checked into the hospital. There is a special budget—co-funded by state authorities—that will cover costs for up to ten days of hospital care. However, when patients who have no medical insurance are not discharged after this ten-day period, the hospital starts to run possibly substantial financial risks. Therefore, the non-event of patients not being discharged after ten days is an important business event that the financial department of the hospital wants to be informed about.

There is an interesting challenge for the Event Delivery Network. It handles events, delivering them to subscribers. But how can it deliver a non-existing event? Where does the non-event come from? Who is responsible for raising the event?

Unlike CEP, the EDN does not detect non-events. We have to implement some logic to find the non-events and turn them into real EDN business events. Here is one way of approaching this challenge: When the patient is admitted to the hospital, an event is published to the Event Delivery Network (say, the PatientAdmittanceEvent) with the patient identifier or a unique key for the admittance record as payload. A composite application is subscribed to this event and a new instance is initiated upon consumption. This composite contains a BPEL component that consumes the PatientAdmittanceEvent. It enters a Pick activity with two branches: One is a Wait activity that will wait for ten days. The other

(Continued)

branch is an onMessage that will wait for a PatientDischargedEvent. The onMessage will correlate this discharge event on the patient or admittance record identifier. When a patient is discharged, the PatientDischargedEvent should be published. When it is, the BPEL component will consume it, conclude the Pick activity, and terminate the instance altogether. However, if after ten days this event is still not received, the wait branch of the Pick activity is activated and it will publish the non-event PatientNotDischargedWithin TenDaysEvent. The BPEL component turns the absence of the business event—after the specified period—into another business event.

The book's wiki has this example described in more detail, including source code.

Summary

Service-Oriented Architecture has been declared dead a couple of times in recent years, partly because it would not bring the level of decoupling organizations are striving for. Some of the criticism of SOA was probably justified. The conclusions, however, were quite over the top. Event-Driven Architecture (EDA), heralded as the successor to SOA, is actually a perfect complement to SOA.

For example, events are a much better way in many cases to "invoke" one-way services. Applications are not burdened with the responsibility to invoke specific services that need to be notified, creating clearly undesirable dependencies, but only need to take on the responsibility to broadcast the occurrence of events to a generic entity.

Thinking about business events and about the business processes, and from there the applications and services that produce them, is a very useful exercise that provides a lot of insight into the workings of the organization. Analyzing those events and specifying canonical definitions for their payloads are the next steps toward implementation of a more decoupled interaction pattern. Once the event definitions have been agreed upon, consumers can start registering—the easy part—and all points of origin of the business events need to be made to publish those events when they occur—the very hard part! Fortunately, we can start small, with a small number of business events and a moderate initial number of event producers. We can add producers as we go—as well as consumers, by the way. An evolutionary introduction of the event-driven way of interacting is quite possible.

The Event Delivery Network in the SOA Suite is the central facilitator that coordinates event definitions and subscriptions on events and also processes the events when they occur—absorbing them and propagating them to all registered subscribers. Mediator, BPMN, and BPEL components can subscribe to events defined in the EDN. Through the EDN, many aspects of Event-Driven Architecture can be implemented in SOA composite applications.