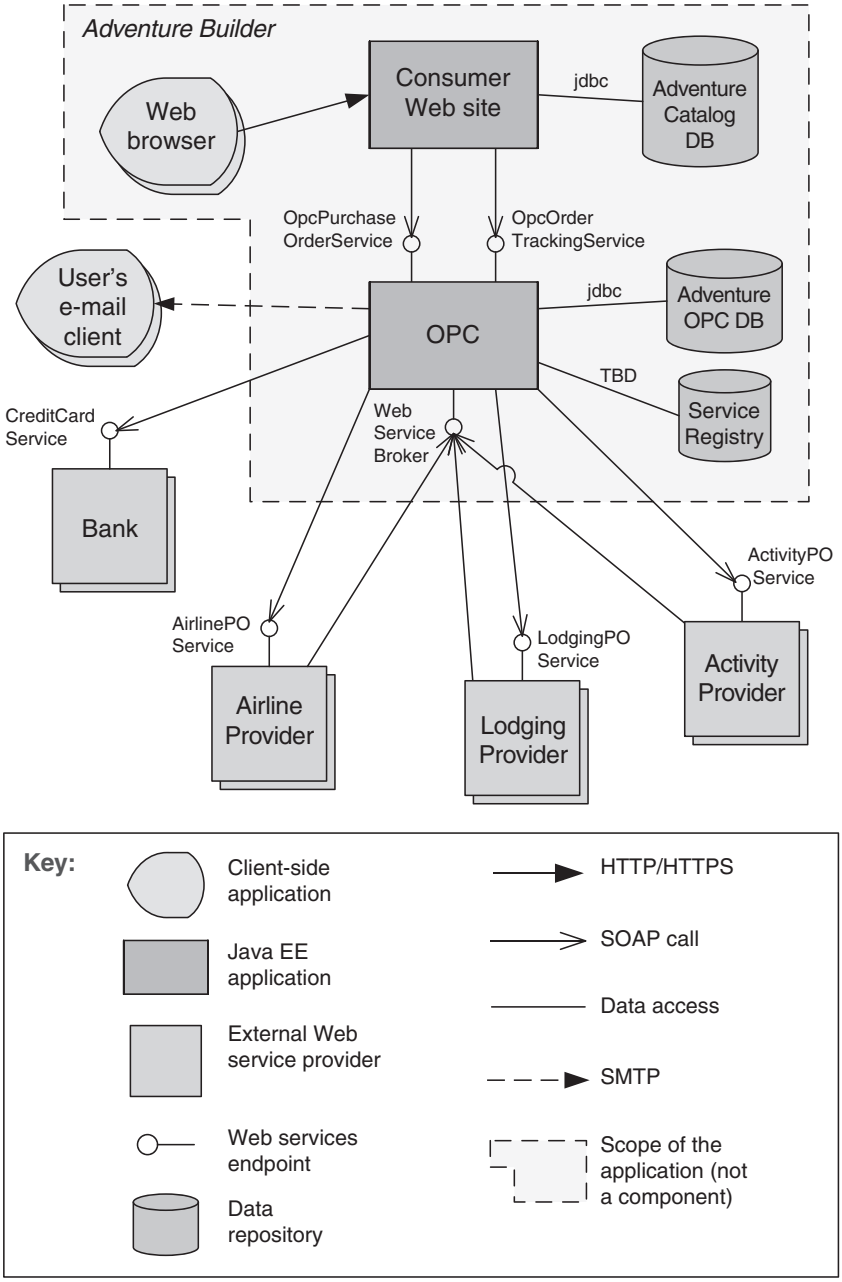


**FIGURE 13.11** Diagram of the SOA view for the Adventure Builder system. OPC stands for "Order Processing Center."

### *Publish-Subscribe Pattern*

**Context:** There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.

**Problem:** How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way that they are unaware of each other's identity, or potentially even their existence?

**Solution:** In the publish-subscribe pattern, summarized in Table 13.8, components interact via announced messages, or events. Components may subscribe to a set of events. It is the job of the publish-subscribe runtime infrastructure to make sure that each published event is delivered to all subscribers of that event. Thus, the main form of connector in these patterns is an *event bus*. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events. Any component may be both a publisher and a subscriber.

Publish-subscribe adds a layer of indirection between senders and receivers. This has a negative effect on latency and potentially scalability, depending on how it is implemented. One would typically not want to use publish-subscribe in a system that had hard real-time deadlines to meet, as it introduces uncertainty in message delivery times.

Also, the publish-subscribe pattern suffers in that it provides less control over ordering of messages, and delivery of messages is not guaranteed (because the sender cannot know if a receiver is listening). This can make the publish-subscribe pattern inappropriate for complex interactions where shared state is critical.

**TABLE 13.8** Publish-Subscribe Pattern Solution

| | |
|---|---|
| Overview | Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers. |
| Elements | *Any C&C component* with at least one publish or subscribe port. Concerns include which events are published and subscribed to, and the granularity of events.<br>*The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events. |
| Relations | The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events. |
| Constraints | All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system.<br>A component may be both a publisher and a subscriber, by having ports of both types. |
| Weaknesses | Typically increases latency and has a negative effect on scalability and predictability of message delivery time.<br>Less control over ordering of messages, and delivery of messages is not guaranteed. |

There are some specific refinements of this pattern that are in common use. We will describe several of these later in this section.

The computational model for the publish-subscribe pattern is best thought of as a system of independent processes or objects, which react to events generated by their environment, and which in turn cause reactions in other components as a side effect of their event announcements. An example of the publish-subscribe pattern, implemented on top of the Eclipse platform, is shown in Figure 13.12.

Typical examples of systems that employ the publish-subscribe pattern are the following:

- Graphical user interfaces, in which a user's low-level input actions are treated as events that are routed to appropriate input handlers
- MVC-based applications, in which view components are notified when the state of a model object changes
- Enterprise resource planning (ERP) systems, which integrate many components, each of which is only interested in a subset of system events
- Extensible programming environments, in which tools are coordinated through events
- Mailing lists, where a set of subscribers can register interest in specific topics

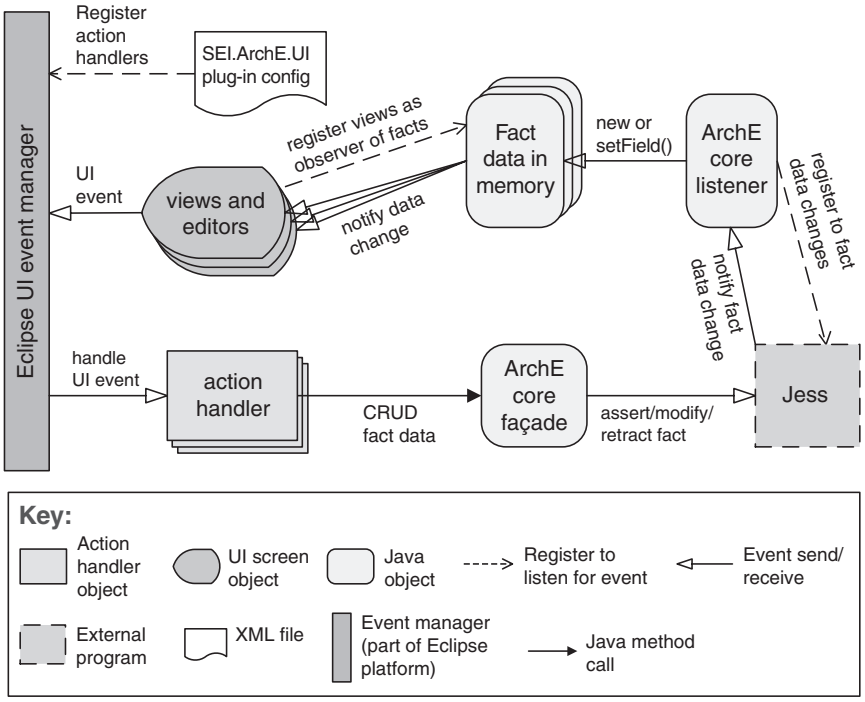


**FIGURE 13.12** A typical publish-subscribe pattern realization

- Social networks, where "friends" are notified when changes occur to a person's website

The publish-subscribe pattern is used to send events and messages to an unknown set of recipients. Because the set of event recipients is unknown to the event producer, the correctness of the producer cannot, in general, depend on those recipients. Thus, new recipients can be added without modification to the producers.

Having components be ignorant of each other's identity results in easy modification of the system (adding or removing producers and consumers of data) but at the cost of runtime performance, because the publish-subscribe infrastructure is a kind of indirection, which adds latency. In addition, if the publish-subscribe connector fails completely, this is a single point of failure for the entire system.

The publish-subscribe pattern can take several forms:

- *List-based publish-subscribe* is a realization of the pattern where every publisher maintains a subscription list—a list of subscribers that have registered an interest in receiving the event. This version of the pattern is less decoupled than others, as we shall see below, and hence it does not provide as much modifiability, but it can be quite efficient in terms of runtime overhead. Also, if the components are distributed, there is no single point of failure.
- *Broadcast-based publish-subscribe* differs from list-based publish-subscribe in that publishers have less (or no) knowledge of the subscribers. Publishers simply publish events, which are then broadcast. Subscribers (or in a distributed system, services that act on behalf of the subscribers) examine each event as it arrives and determine whether the published event is of interest. This version has the potential to be very inefficient if there are lots of messages and most messages are not of interest to a particular subscriber.
- *Content-based publish-subscribe* is distinguished from the previous two variants, which are broadly categorized as "topic-based." Topics are predefined events, or messages, and a component subscribes to all events within the topic. Content, on the other hand, is much more general. Each event is associated with a set of attributes and is delivered to a subscriber only if those attributes match subscriber-defined patterns.

In practice the publish-subscribe pattern is typically realized by some form of message-oriented middleware, where the middleware is realized as a broker, managing the connections and channels of information between producers and consumers. This middleware is often responsible for the transformation of messages (or message protocols), in addition to routing and sometimes storing the messages. Thus the publish-subscribe pattern inherits the strengths and weaknesses of the broker pattern.