David Chappell
& Associates

# INTRODUCING SCA

DAVID CHAPPELL

JULY 2007

## CONTENTS

## SCA FUNDAMENTALS

What is an application? One way to think of it is as a set of software components working together. All of these components might be built using the same technology, or they might use different technologies. They might run inside the same operating system process, in different processes on the same machine, or across two or more connected machines. However an application is organized, two things are required: a way to create components and a mechanism for describing how those components work together.

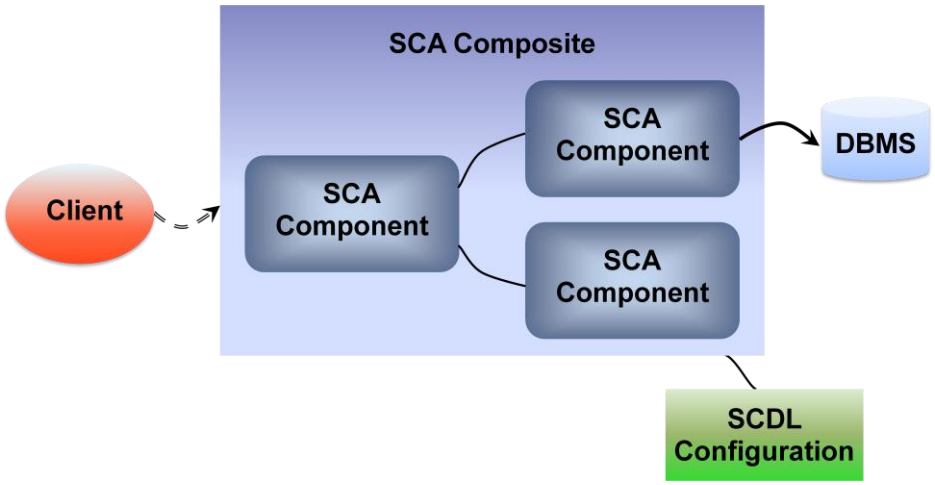
Service Component Architecture (SCA) defines a general approach to doing both of these things. Now owned by OASIS, SCA was originally created by a group of vendors, including BEA, IBM, Oracle, SAP, and others. The SCA specifications define how to create components and how to combine those components into complete applications. The components in an SCA application might be built with Java or other languages using SCA-defined programming models, or they might be built using other technologies, such as the Business Process Execution Language (BPEL) or the Spring Framework. Whatever component technology is used, SCA defines a common assembly mechanism to specify how those components are combined into applications.

This overview provides an architectural introduction to SCA. The goal is to provide a big-picture view of what this technology offers, describe how it works, and show how its various pieces fit together.

### COMPONENTS AND COMPOSITES

Every SCA application is built from one or more components. In a simple SCA application, the components could be Java classes running in a single process, and their interactions might rely on Java interfaces exposed by those classes. In a slightly more complex case, the Java classes in this application might be running on different machines, relying on some communication mechanism to interact with one another. In a still more complex case, the application might contain a few components implemented as Java classes, others written in C++, and still others defined using BPEL, all spread across a group of machines. In all of these situations, the same fundamental issues exist: There must be a way to define components and to describe how they interact. And in an increasingly service-oriented world, those interactions should be modeled as services.

To do this, SCA provides a generalized definition of a component. It also specifies how those components can be combined into larger structures called *composites*. The figure below shows how a simple composite built from three SCA components might look.




A composite is a logical construct: Its components can run in a single process on a single computer or be distributed across multiple processes on multiple computers. A complete application might be constructed from just one composite, as in the example shown here, or it could combine several different composites. The components making up each composite might all use the same technology, or they might be built using different technologies—either option is possible.

As the figure shows, an SCA application can be accessed by software from the non-SCA world, such as a JavaServer Page (JSP), a Web services client, or anything else. Components in an SCA application can also access data, just like any other application. One option for this is to use Service Data Objects (SDO), perhaps in concert with a standard Java data access technology such as JDBC or Java EE 5's Java Persistence API (JPA). An SCA component can also use JDBC, JPA, or something else directly—the SCA specifications don't mandate any particular choice.

An SCA composite is typically described in an associated configuration file, the name of which ends in *.composite*. This file uses an XML-based format called the *Service Component Definition Language (SCDL*, commonly pronounced "skiddle") to describe the components this composite contains and specify how they relate to one another. For the three-component composite shown above, the basic structure of its SCDL configuration would look like this:

```
<composite name="ExampleComposite" ...>

   <component name="Component1">
      ...
   </component>

   <component name="Component2">
      ...
   </component>

   <component name="Component3">
      ...
   </component>

</composite>
```

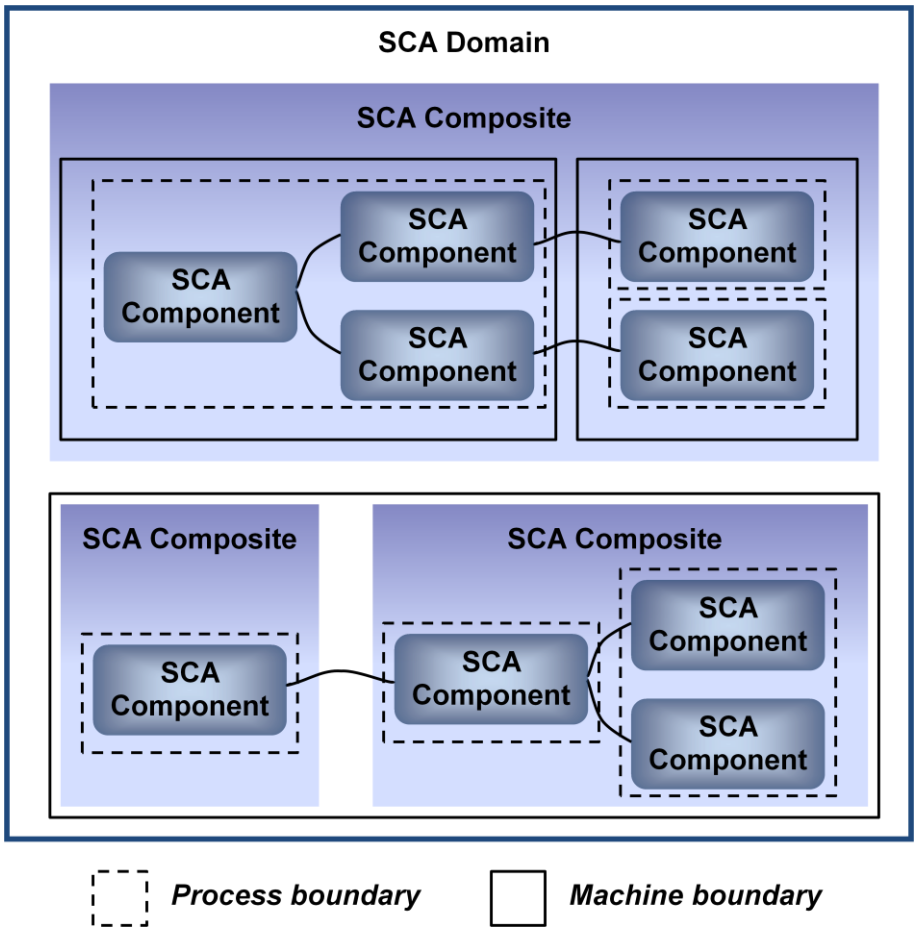
Components and composites are the fundamental elements of every SCA application. Both are contained within a larger construct called a *domain*, however, and so understanding SCA requires understanding domains. This fundamental idea is described next.

## DOMAINS

An implicit assumption of SCA's creators was that a given environment would install a group of SCA products, commonly known as *runtimes*, from a single vendor. For example, suppose a division of a large firm chooses a particular company as its SCA vendor. This division is likely to install their chosen vendor's SCA runtime on a number of machines. This isn't an unreasonable expectation, as it mirrors how organizations have typically purchased and installed J2EE products. These SCA runtimes will likely be managed by the same group of people, and this set of systems—with a common vendor's runtime technology and common management—provides the primary example of a domain.

Domains are an important concept in SCA. To see why, realize that even though SCA allows creating distributed applications, it doesn't fully define how components on different machines should interact. As a result, the communication among these components will be implemented differently by different products. (As described in the section *Implementing SCA* below, however, an SCA runtime can allow a third party to create a *container* that plugs into that runtime to support a particular technology, such as BPEL.)

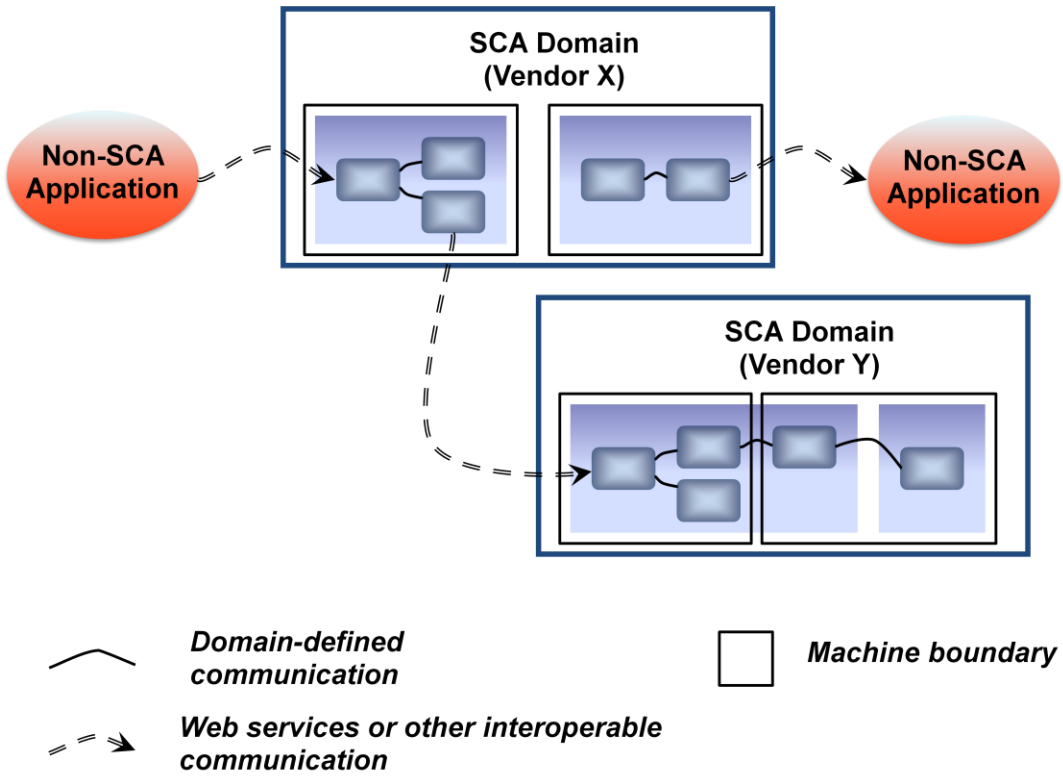
A domain can contain one or more composites, each of which has components implemented in one or more processes running on one or more machines. The figure below shows an example of how this might look.




The domain shown here contains three composites and three computers. One composite, shown in the upper part of the figure, consists of five components spread across three processes in two different machines. The other two composites, shown in the lower part of the figure, run all of their components on a single machine, dividing them into three separate processes. How communication happens between these components, whether it's intra-process, inter-process, or inter-machine, can be defined differently by each SCA vendor. Whatever choice is made, composites are single-vendor constructs—they don't span domain boundaries.

It might seem odd for a multi-vendor specification to define a way to create distributed applications, yet not define how the components in those applications interact. To understand this, realize that the primary goal of SCA's creators was to allow portability of code and developer skills across different SCA implementations. While creating composites that span domains—and thus vendor boundaries—might one day be possible, this wasn't a goal for the first version of SCA. Also, limiting composites to a single domain allows useful optimizations. An SCA developer's life is significantly simpler inside a domain, for example, since the complexities inherent in configuring multi-vendor applications can be avoided.

Yet don't be confused. Even though an SCA composite runs in a single-vendor environment, it can still communicate with applications outside its own domain. To do this, an SCA component can make itself accessible using an interoperable protocol such as Web services. The figure below shows how this looks.




This example shows two SCA domains, each with two computers. One domain uses vendor X's SCA runtime, while the other uses vendor Y's SCA runtime. All of the communication between components and composites within each domain is done in a vendor-specific way—SCA doesn't mandate how this interaction should happen. To communicate between domains, however, or with non-SCA applications, a component will typically allow access via Web services or some other interoperable mechanism. In fact, an SCA application communicating with another SCA application in a different domain sees that application just like a non-SCA application; its use of SCA isn't visible outside its domain.

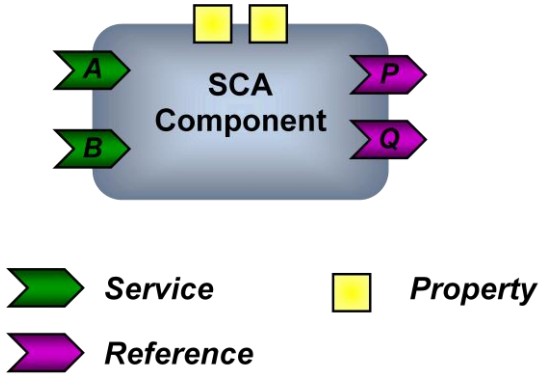
## UNDERSTANDING COMPONENTS

Components are the atoms from which an SCA application is created. Like atoms, SCA components behave in consistent ways, and they can be assembled into different configurations. Understanding SCA starts with understanding these fundamental application building blocks.

In the parlance of SCA, a component is an instance of an *implementation* that has been appropriately configured. The implementation is the code that actually provides the component's functions, such as a Java class or a BPEL process. The configuration, expressed in SCDL, defines how that component interacts with the outside world. In theory, an SCA component could be implemented using pretty much any technology. Yet whatever technology is used, every component relies on a common set of abstractions, including services, references, properties, and bindings, to specify its interactions with the world outside itself. This section describes each of these.

## SERVICES, REFERENCES, AND PROPERTIES

Looked at from the outside, an SCA component is a simple thing. Whatever technology is used to create it, every component has the same fundamental parts, as shown below.




Each component typically implements some business logic, exposed as one or more *services*. A service, represented by a green chevron in the figure, provides some number of *operations* that can be accessed by the component's client. How services are described depends on the technology that's used to implement the component. A Java component, for example, might describe its services using ordinary Java interfaces, while a component implemented in BPEL would likely describe its services using the Web Services Description Language (WSDL).

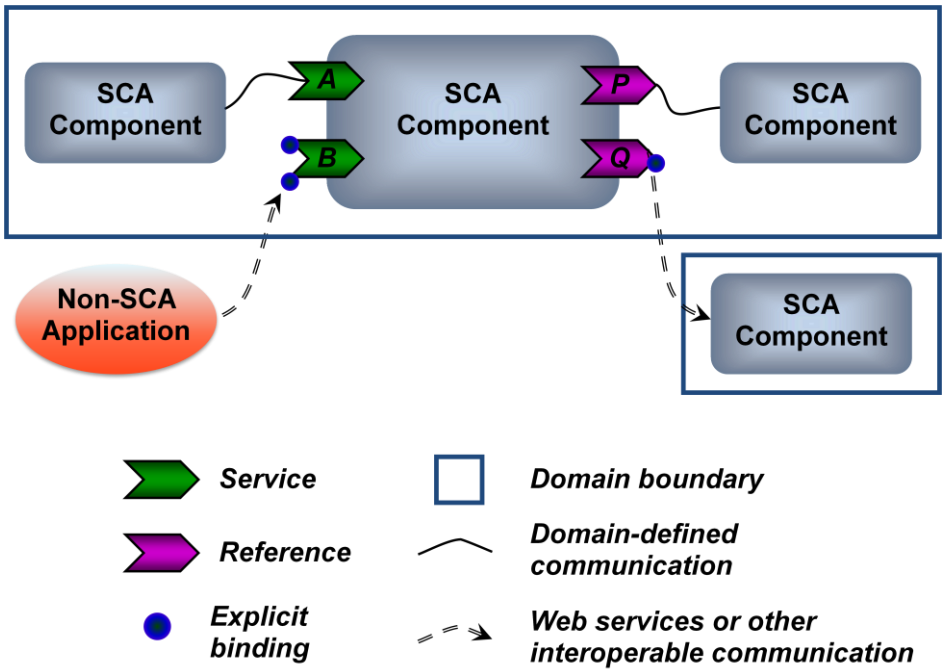
Along with providing services to its own clients, a component might also rely on services provided by other components in its domain or by software outside its domain. To describe this, a component can indicate the services it relies on using *references*. Shown as a purple chevron in the figure above, each reference defines an interface containing operations that this component needs to invoke.

These core ideas of services and references are worth lingering over for a moment. It's become common to use services to model what a component provides to its clients. Rather than the distributed object approach of the 1990s, the slightly less coupled approach of services now appears to be a better choice. Explicitly defining references has become popular more recently, and it offers several advantages. For one thing, formally expressing a component's dependencies can help make relationships among chunks of code clearer to developers, something that's always welcome. Explicit references also allow what's sometimes known as *dependency injection*. This opaque phrase actually has a simple meaning: Instead of requiring a developer to write code that locates the service a component depends on, the SCA runtime can locate that service for her. Less code is good, as is the ability to move components more easily from one environment to another without needing to change any lookup code they contain.

Along with services and references, a component can also define one or more *properties*. Each property contains a value that can be read by that component from the SCDL configuration file when it's instantiated. For example, a component might rely on a property to tell it what part of the world it's running in, letting it customize its behavior appropriately.

## BINDINGS

Services and references let a component communicate with other software. By design, however, they say nothing about how that communication happens. Specifying this is the job of *bindings*. The figure below shows where bindings fit into the SCA picture.




A binding specifies exactly how communication should be done between an SCA component and something else. Depending on what it's communicating with, a component might or might not have explicitly specified bindings. As the figure shows, a component that communicates with another component in the same domain, even one in another process or on another machine, need not have any explicit bindings specified. Instead, the runtime determines what bindings to use, freeing the developer from this chore.

To communicate outside its domain, however, whether to a non-SCA application or an SCA application running in some other domain, a component's creator must specify one or more bindings for this communication. Each binding defines a particular protocol that can be used to communicate with this service or reference. A single service or reference can have multiple bindings, allowing different remote software to communicate with it in different ways.

Because bindings separate how a component communicates from what it does, they let the component's business logic be largely divorced from the details of communication. This is a departure from the approach taken by older technologies, which tended to mix the two. Separating these independent concerns can make life simpler for application designers and developers.

## AN EXAMPLE: SCA'S JAVA COMPONENT MODEL

The fundamental abstractions of an SCA component are simple: services, references, properties, and (sometimes) bindings. Abstractions aren't enough, however. There must also be a way to create components that implement these abstractions.

Some existing technologies already match well with the abstractions of an SCA component. For example, the Spring Framework provides explicit support for services, references, and properties, and so mapping these into SCA's similar concepts is straightforward. Because of this, the specification defining how to create SCA components using Spring is only a few pages long. Similarly, BPEL also provides some built-in support for the abstractions of an SCA component. BPEL's concept of partnerLinks, for example, can be mapped to both services and references. While extensions are required for using properties, SCA's specification for creating components using BPEL is quite short, no more than a dozen pages.

Yet even though BPEL and Spring are viable options for creating SCA components, neither was created with SCA in mind. Given this, why not design a programming model from the ground up that's explicitly intended for building SCA components? This is exactly what's done by SCA's Java component model. The next section describes how SCA components can be created using this new programming model.

Before doing this, it's worth thinking about why SCA's creators chose to invent yet another new component model for Java. One important motivation was the need for an explicitly service-oriented approach. The current Java programming models for business logic, such as Enterprise JavaBeans (EJB), were defined for an earlier world where services weren't viewed as fundamental. Accordingly, none of the Java EE 5 technologies were designed to match SCA's view of components. Also, because bindings separate communication details from business logic, an SCA-based Java component model can support diverse communication styles in a common way. For both of these reasons, using SCA's new component model can significantly simplify a Java developer's life.

### Defining Services

Unlike the older J2EE technologies, SCA's Java programming model relies on annotations rather than API calls. This approach makes creating a basic service quite easy. In fact, for a service with local clients, nothing at all is required: an ordinary Java interface and class will do. A service that's accessible by remote clients, however, must indicate this fact by marking an interface with an appropriate annotation, as this simple example shows:

```
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface AS
{
      int add(int a, int b);
      int subtract(int a, int b);
}

public interface MD
{
      int multiply(int a, int b);
      int divide(int a, int b);
```

```
}

public class Calculator implements AS, MD {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    public int divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException();
        } else {
            return a / b;
        }
    }
}
```

This example begins by importing an annotation definition from a standard SCA package. It then uses this annotation, **@Remotable**, to indicate that the service provided by the **AS** interface can be made accessible to remote clients. While there's a bit more that needs to be defined for this component in the SCDL configuration, as described later, this annotation is all that's required in the Java code. The SCA runtime does everything required to make the service accessible to remote clients. This simple component also provides a second service that exposes the operations defined in the **MD** interface. Because this service is accessible only by local clients, nothing extra is required, and so this interface has no annotations.

Both interfaces are implemented by the same class, here given the unoriginal name **Calculator**. Because they're defined in an interface marked with **@Remotable**, the **Add** and **Subtract** methods can be invoked by either local or remote clients. The **Multiply** and **Divide** methods, defined in an interface without the **@Remotable** annotation, can be called only by clients running in the same process as an instance of the **Calculator** class.

## Defining References

Services let a component describe what it provides to the world outside its boundaries. References let a component express what it needs from that world. In SCA's Java programming model, references are specified using the **@Reference** annotation. For instance, suppose the example calculator just described depends on a monitoring service to keep track of its usage. A reference to that service might be defined like this:

```
@Reference
protected MonitorService monitorService;
```

**MonitorService** is an interface, and so the component can invoke methods in this interface in the usual way. To invoke, say, a **usageCount** method, the component could just call

```
monitorService.usageCount(x);
```

Yet the component never needs to create an instance of a class that implements the **MonitorService** interface. Instead, the runtime automatically locates a component that provides this interface, then sets the value of **monitorService** to point to that service. Rather than relying on the developer to write code that finds the service, this responsibility is passed to the runtime. (Although this approach is most often called dependency injection, it's also sometimes referred to as *inversion of control*.)

The details of how a runtime finds an instance of a service that satisfies this reference are domain-specific; how it happens is left up to the creator of each SCA runtime. Because of this, don't expect that references can be automatically linked to services provided by components in another SCA domain. Within a single-vendor environment, however, using references can simplify a developer's life.

## Defining Properties

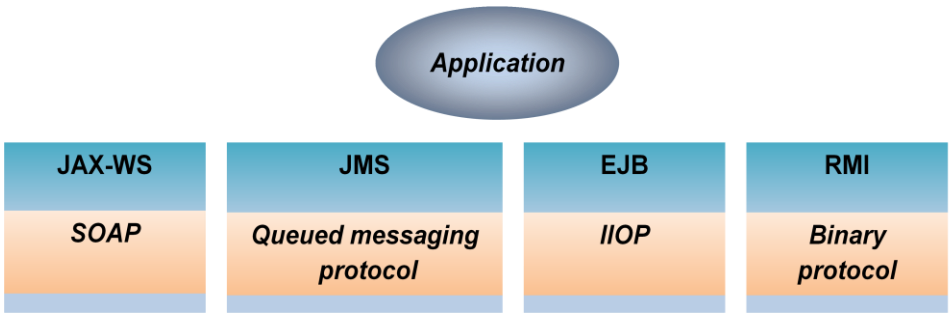
Properties are a simple idea, and so using them in Java is also simple. Here's an example:

```
@Property
protected String region;
```

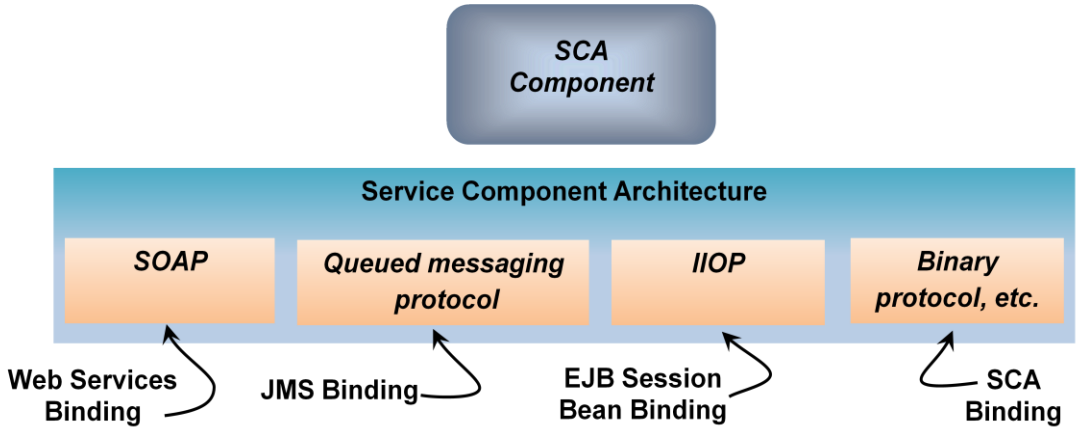
Like references and remote services, properties are identified using an annotation: **@Property**. This annotation can be assigned to a field in a Java class or to a setter method, and in either case, it indicates that a value should be read from the SCDL configuration file of the composite to which this component belongs. Properties can also be more complex—they needn't be just single-valued strings or integers or other simple types. Whether they're simple or complex, however, the goal is the same: providing a way to configure a component via values that are read at runtime.

## Defining Bindings

As described earlier, bindings determine how a component communicates with the world outside its domain. Bindings can be assigned to services and to references, and each one specifies a particular protocol. To illustrate why bindings are useful, think of how applications use different protocols in Java EE5 and its J2EE predecessors. As shown below, each protocol is provided by a distinct technology, so each one has its own application programming interface. Using SOAP over HTTP, for example, typically means building on JAX-WS (or JAX-RPC in J2EE 1.4), while using a queued messaging protocol requires the Java Message Service (JMS). This forces developers to learn different APIs, perhaps with entirely different programming models, to use different protocols. It also mixes business logic with communication code, further complicating a developer's life.



SCA takes a simpler approach. Rather than wrapping different protocols into distinct technologies with different APIs, SCA allows each remotable service and each reference to specify the protocols it supports using bindings. The programming model seen by an application remains the same regardless of which protocol is used, as the figure below illustrates.




To be accessible via SOAP over HTTP, for example, an SCA service uses the Web Services binding, while access via a queued messaging protocol uses the JMS binding. Similarly, the EJB session bean binding allows access to session beans using the Internet Inter-ORB Protocol (IIOP). Every SCA runtime also provides an *SCA binding*. The protocol this binding uses isn't specified, however. Instead, the SCA binding is only used when a service and its client are both running in the same domain. Since every vendor wants applications built on its products to perform as well as possible, it's safe to assume that this binding will most often use a binary protocol. This isn't required, however; an SCA runtime is free to choose different protocols in different situations, all of which fall under the umbrella of the SCA binding.

Version 1.0 of the SCA Java component model defines no way for a developer to specify a binding directly in Java. Instead, the bindings a service or reference relies on are either chosen by the runtime, for intra-domain communication, or set explicitly in a component's SCDL configuration file. Here's an example of how a binding for a component's service might be specified:

```
<binding.ws uri="http://www.qwickbank.com/services/serviceA"/>
```

This example **`binding`** element specifies two things: what protocol the binding uses and where the service can be accessed using this protocol. The **`.ws`** in the element's name indicates the first of these, specifying the Web Services binding. The element's **`uri`** attribute indicates the second, specifying the URL at which the service can be found. (It's also possible—and much more likely—for a Web services binding to use a relative URL rather than the absolute form shown here.) Other bindings can be specified in a similar way. The **`binding.jms`** element specifies the JMS binding, for example, while **`binding.ejb`** indicates the EJB session bean binding.

## Defining Other Aspects of a Component

Along with the **`@Remotable`** attribute shown earlier, the SCA Java component model defines a number of others. Among the most important of these are the following:

- **`@OneWay`**, specifying that an operation returns no response and so doesn't block waiting for one.
- **`@Scope`**, controlling the component's lifetime. For example, a component can be *conversational,* which means that it maintains its state between method calls, or *stateless*, maintaining nothing between calls.
- **`@Callback`**, allowing a callback interface to be defined. This supports two-way communication between components using what SCA calls *bi-directional* interfaces.

Not all attributes are usable with all bindings. For instance, the **`@Scope`** attribute with the conversational option can only be used with protocols that can pass session information, such as a SOAP binding using WS-ReliableMessaging. As with any programming environment, SCA developers must understand their technology to use it correctly.

## CONFIGURING A COMPONENT

Whether it's implemented using SCA's Java component model or another technology, every SCA component relies on information in the SCDL configuration file associated with the composite it belongs to. As shown earlier, each component is defined using the **`component`** element, and components are contained within a **`composite`** element. Exactly what must be specified for a component depends on whether it's defining communication with other components in the same SCA domain or with software outside its domain. In the simple (and probably more common) case, where a component interacts only with other components in the same domain, its **`component`** element can be quite straightforward. For the **`Calculator`** class shown earlier, that element might look like this:

```
<component name="Component1">
   <implementation.java class="services.examples.Calculator"/>
   <property name="region">
      Europe
   </property>
</component>
```

Like all **`component`** elements, this one assigns the component a name and provides a wrapper for other elements. The first of these, **`implementation.java`**, indicates that this component is implemented using the SCA Java component model, then specifies the Java class in which this implementation can be found. The second element, **`property`**, defines a value for the component's property. Whatever value is provided is read into the **`region`** field in this component when it begins executing. Note that neither services nor references for this component need be described here. Instead, the runtime can discover these things by introspection—there's no requirement to list them explicitly. And because all communication is happening within the same domain, the runtime can choose which bindings to use, obviating the need to specify them here.

If the **`Calculator`** class is communicating outside its domain, however, things get slightly more complex. Suppose, for instance, that both its remotable service and its reference can be connected to software outside this component's domain. In this case, the **`component`** element for the class might look like this:

```
<component name="Component1">
   <implementation.java class="services.examples.Calculator"/>
```

```
   <service name="AS">
      <binding.ws uri="http://www.qwickbank.com/services/serviceA"/>
   </service>
   <reference name="MonitorService">
      <binding.ws uri="http://www.qwickbank.com/services/serviceM"/>
   </reference>
   <property name="region">
      Europe
   </property>
</component>
```

Just as before, the component's description begins with an **implementation.java** element indicating what technology was used to implement the component and where this implementation can be found. It also ends with the **property** element as before. In between, however, are explicit **service** and **reference** elements for the remotable service and the reference this component defines. Each of these specifies the Web services binding, complete with a URL. Because the component is communicating with software outside its domain, the runtime can't choose a binding. Instead, the component explicitly specifies that an interoperable binding should be used. (Note that this isn't required for the component's local service, since it's only accessible from within the same domain.) It's up to the SCA runtime to generate WSDL interfaces from the Java interfaces, fix up the service to be callable via SOAP, and do everything else required to let this component communicate via Web services.

As described here, the Calculator component is implemented using SCA's Java component model. If some other technology were used to implement it, however, its definition in the SCDL configuration wouldn't change much. If this component were implemented in BPEL, for example, and communicated only with other components in its own domain, its component element might now look like this:

```
<component name="Component1">
   <implementation.bpel process="ExampleProcess"/>
   <property name="region">
      Europe
   </property>
</component>
```

Rather than the **implementation.java** element shown earlier, a BPEL component uses the **implementation.bpel** element, naming a BPEL process rather than a Java class. Nothing else need change. While the runtime must behave differently to execute this BPEL component, SCA's abstract component definition remains the same, and so only small changes are required in the SCDL configuration file.

Similarly, a component built using the Spring Framework would specify the **implementation.spring** element, while one built using SCA's C++ component model would use **implementation.cpp**. An entire composite can also act as a component in another composite, an option that relies on the **implementation.composite** element. This approach allows composites to be nested, regardless of the technologies from which their components are built.

## UNDERSTANDING COMPOSITES

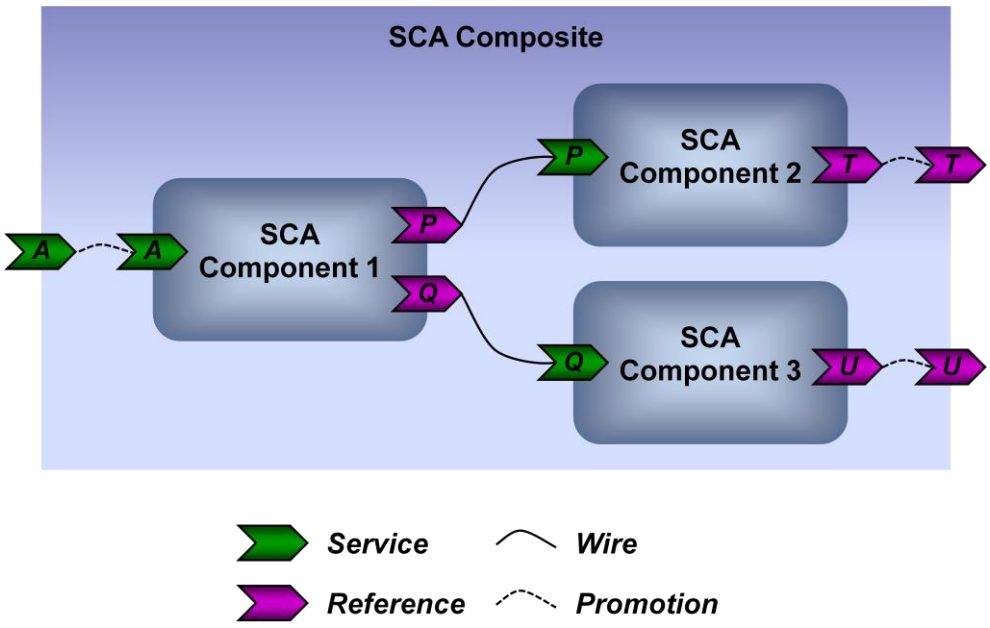
If components are the atoms of SCA, then composites are the molecules. Composites group components into useful combinations, which can themselves be further combined. This building-block approach to

creating applications has some obvious pluses. For example, providing a well-defined set of abstractions for components can help people who create applications think more clearly about how those applications should be designed. Keeping these abstractions consistent across different technologies also makes building applications using different languages and runtimes easier. Recall, too, that the components in a composite might run in the same process, in different processes on a single machine, or in different processes on different machines. In all of these cases, it's useful to have some way to deploy the entire application as a unit. And since components provide discrete, well-defined services, a graphical tool could allow assembling or re-assembling various components as needed to address a particular problem. Doing this can make a developer's life easier, and it might even allow less technically adept people to create applications by assembling existing components.

Achieving these goals requires defining how components relate to one another within a composite, relationships described in SCA's assembly model specification. This section takes a closer look at how an SCA composite is assembled.

## WIRES AND PROMOTION

As usual, it's useful to start with a picture. The figure below shows three components, each with some combination of services and resources. All three are part of the same composite.




As the figure shows, a reference in one component is connected to a service in another component using a *wire*. A wire is an abstract representation of the relationship between a reference and some service that meets the needs of that reference. Exactly what kind of communication a wire provides can vary—it depends on the specific runtime that's used, what bindings are specified (if any), and other things. And since the components in a composite might run entirely within a single process, across processes on a single machine, or be spread across processes on different machines, wires can represent relationships in all of these cases.

Just as components expose services, a composite can also expose one or more services. These services are actually implemented by components within the composite. To make them visible to the outside world, the composite's creator can *promote* those services. In this example, service A implemented by Component 1 is promoted to be a service provided by the composite itself. Similarly, zero, one, or more references defined by components can be promoted to be visible outside the composite. Here, both references T and U are promoted to the composite level.

## CONFIGURING A COMPOSITE

All of the relationships in a composite are expressed in the SCDL configuration file. Here's a slightly simplified example of how this file might look for the scenario shown above:

```
<composite name="ThreeComponents" autowire="true"...>

   <component name="Component1">
      <implementation.bpel process="Process1"/>
   </component>

   <component name="Component2">
      <implementation.java class="services.examples.class2"/>
   </component>

   <component name="Component3">
      <implementation.java class="services.examples.class3"/>
   </component>

   <service name="A" promote="Component1/A"
      <binding.ws/>
   <service/>

   <reference name="T" promote="Component2/T"/>
   <reference name="U" promote="Component3/U"/>

</composite>
```

Like all SCDL configurations, this one wraps its contents in a **composite** element. In the example shown here, this element's **autowire** attribute is set to true. This indicates that the SCA runtime should automatically attempt to connect the services and references defined by the components in this composite. To do this, the runtime looks for matches between the references and services exposed by this composite's components. To be a match, a service must provide the interface a reference requires, allow using a compatible binding, and perhaps more. Although it's not shown here, it's also possible to define explicit wires between components using a **wire** element.

Next appear **component** elements describing each of the three components in this composite. The first component is implemented in BPEL, as indicated by the **implementation.bpel** element. The other two components are implemented using SCA's Java component model, and so both use the **implementation.java** element instead. Even though each component has the services and references shown in the diagram, none are explicitly specified in these **component** elements. Instead, the runtime can discover them and choose appropriate bindings, as described earlier.

After all three components have been defined, the service provided by the composite itself is specified using the **`service`** element. This example promotes service A in Component 1 to be a visible service of this composite. This service is meant to be accessible from outside the composite's domain, and so it defines an explicit Web services binding. In this example, no URL is specified—the runtime can supply one once the application is deployed. Similarly, each of the two **`reference`** elements that end this example promotes a reference from one of the composite's components, making those references visible outside the composite itself. The example assumes that only other components within this domain are referenced, and so their bindings need not be specified.

Building modern enterprise applications is unavoidably complex. As more technologies are used to implement business logic, such as BPEL, Spring, and SCA's Java component model, that complexity increases. A primary goal of SCA composites is to provide a consistent way to assemble these different technologies into coherent applications and so make this diversity more manageable.

## USING POLICY

Interactions between the parts of a distributed application can get complicated. One way to make things more manageable is to let developers use *policies* to specify their intent—what they want to happen—then let something else figure out how to achieve this intent. To support this, SCA defines a policy framework.

This framework defines two broad categories of policies:

- Interaction policies: Modify how a component interacts with other components. Examples include policies that define requirements for security or for reliable message transfer. Interaction policies are typically applied to bindings.
- Implementation policies: Modify how a component behaves locally. This kind of policy might specify that a component must run inside a transaction, for example (although the initial 1.0 version of the SCA specs doesn't define transactional policies).

Like many other things in SCA, policies can be declared in a SCDL configuration file. For Java SCA components, policies can also be defined using annotations assigned to interfaces, methods, and other things. For example, the annotation **`@Confidentiality`** indicates that communication should be confidential (that is, encrypted), while **`@Authentication`** indicates that authentication is required.

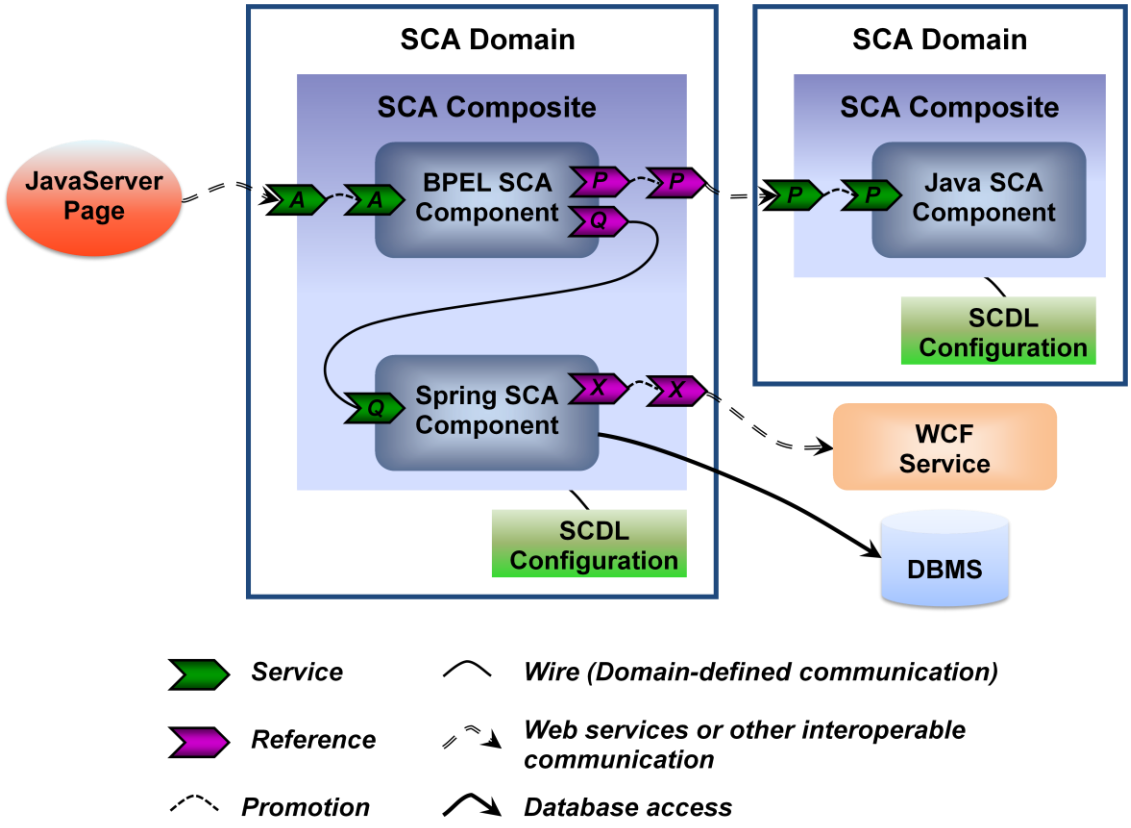
Yet what exactly do these annotations mean? The answer depends on how each of these policies is defined within the domain in which this SCA component is running. To define policies, SCA posits a *policy administrator* role in each domain. This administrator specifies what a particular policy means in her domain by specifying *intents* and *policySets*, each of which contains one or more policies. For example, a binding for a service can have an associated policySet describing its interaction policies, while a binding for a reference can have another policySet describing its interaction policies. When a wire is created between them, these policySets are matched, and their intersection determines the set of policies used for this communication.

SCA doesn't define how policies should be described within a domain—no one policy language is mandated—and so each vendor is free to do this in any way it likes. Between domains, however, where communication is likely to rely on Web services, policies can be specified in a vendor-neutral form using

WS-Policy. And because policies are defined at the domain level, it's possible that policy requirements could sometimes influence domain boundaries. As described earlier, a domain will typically consist of a set of SCA runtimes provided by a single vendor and managed by a single group. Yet even within this environment, different parts of an organization might require different policies. Suppose, for example, that two departments in the same company use the same vendor's SCA product but have different security requirements. To address this, the firm might choose to create two separate SCA domains, each with distinct security policies.

## PUTTING THE PIECES TOGETHER: ILLUSTRATING AN SCA APPLICATION

SCA defines a general framework for creating applications. The best way to come to grips with this generality is to illustrate a representative example. The figure below shows how an application created using SCA might look.




In this example, the client is a JavaServer Page. This JSP invokes service A, which is provided by an SCA component that's part of a simple composite in some SCA domain. This component is implemented in BPEL, and its service is promoted to be visible outside the composite, a fact that's expressed in the SCDL configuration file.

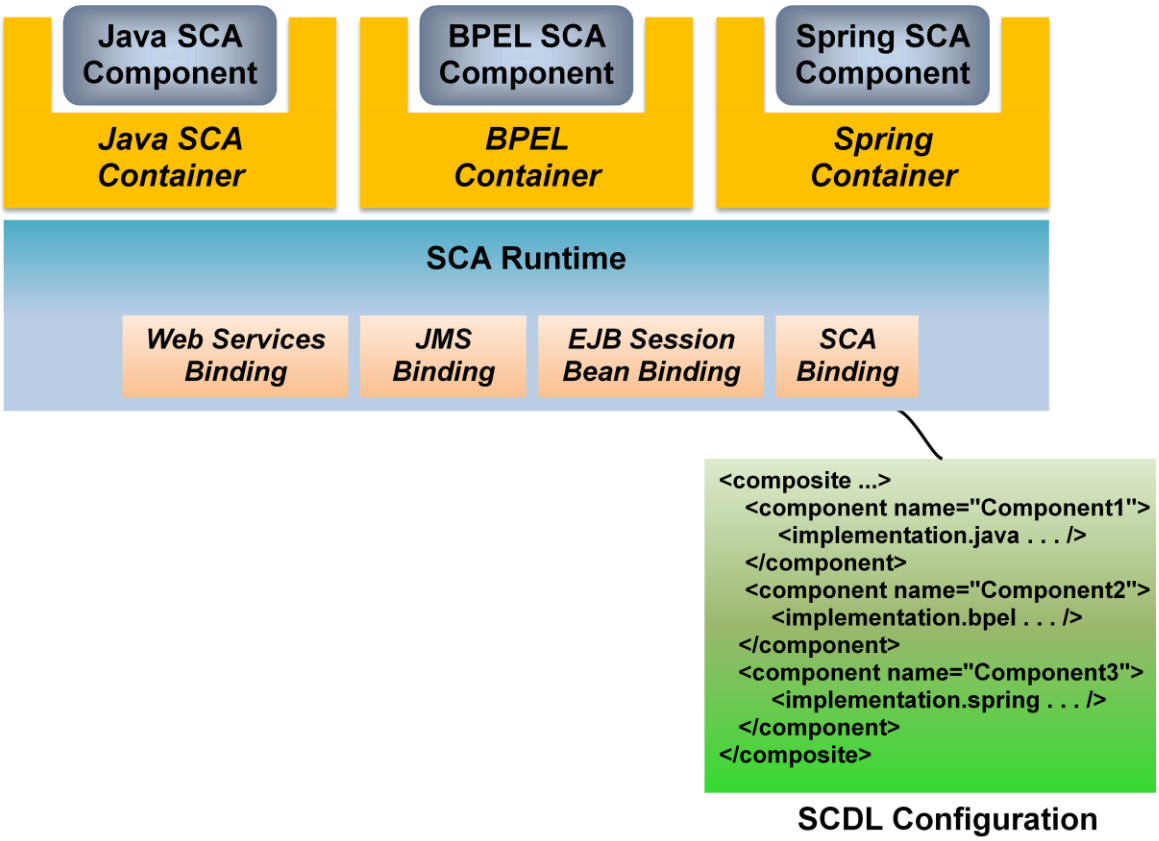
This BPEL component contains references to two other services, P and Q. Service P is provided by a component built with SCA's Java component model, a component that's part of another composite in another SCA domain. Accordingly, communication with this component relies on Web services (or perhaps some other interoperable protocol). Service Q is implemented by a Spring SCA component that's

part of the same composite as the BPEL component. Communication between the BPEL component and the Spring component relies on the domain-specific SCA binding, modified by whatever policies have been specified.

While the Spring component implements service Q, it relies on service X. This service is provided by an application created with Microsoft's Windows Communication Foundation (WCF)—it's not SCA-based—and so communication once again relies on Web services. It's worth reiterating that communication with a non-SCA service looks just like communication with an SCA-based service in another domain. Internal implementation details aren't visible in either case, and so both appear as ordinary Web services. Finally, the Spring component accesses a database, as the figure shows. This access can be done through SDO or directly with JPA or JDBC or another data access technology.

## IMPLEMENTING SCA

The SCA specs say essentially nothing about how this technology should be implemented. Two open source implementations exist today—Tuscany and Fabric3—and various vendors are creating their own implementations as well. While it's not required, the diagram below illustrates a common approach to building an SCA runtime.




As shown here, an SCA runtime might provide a number of *containers*, one for each component technology it supports. The SCDL configuration file's `implementation` element tells the runtime both what kinds of containers it needs for a particular composite and where to find the implementations of the

components in that composite. In this example, for instance, the SCDL configuration file specifies that containers implementing SCA's Java component model, BPEL, and Spring are required. The SCA runtime provides all bindings, allowing components created using any technology to use any available binding. The runtime also defines all communication between components, regardless of how those components are implemented.

Although it's not required, an SCA implementer can make public the interface between its runtime and containers. Doing this allows third parties to create their own containers, making the SCA runtime more extensible. One option for this interface is Java Business Integration (JBI), a specification created under the Java Community Process. JBI supports only Java-based technologies, however—using it for, say, a C++ container would be problematic—and it hasn't received wide support from the major Java vendors. While some SCA vendors might choose to use it, neither Tuscany nor Fabric3 implement JBI.

It's also possible for the creator of an SCA runtime to use OSGi. Created by a multi-vendor working group, this specification has received significantly more support from the major SCA vendors than has JBI. OSGi defines how to package code into bundles, and while it's also Java-only, some creators of SCA runtimes might allow extensions such as a new binding to be added as an OSGi bundle.

Runtimes and containers are certainly necessary, but so are development tools. SCA tools are being provided by various vendors, and there's also an open source tooling project underway. Known as the SOA Tools Platform Project, one of its goals is to create SCA-based development tools for Eclipse.

## CONCLUSION

Different people look at SCA in different ways. The specifications offer plenty of options, and so when someone says "SCA", he might mean any or all of the things these specs define. Similarly, different vendors are almost certain to emphasize different aspects of SCA. One vendor might support SCA's assembly aspects and its new programming model for Java components, for example, but not the C++ version of this model. Another might support only SCA's assembly aspects, completely ignoring the new Java and C++ programming models. And since these specifications explicitly allow vendor extensions, look for each vendor to provide some customization in its SCA products.

Still, SCA is unquestionably an interesting technology. By providing an alternative to older approaches such as EJB and JAX-WS, it can offer a new way to create Java business logic for a service-oriented world. By providing an assembly mechanism for components implemented using various technologies, it can help knit together an increasingly diverse environment.

The group of vendors backing SCA is large and impressive. Plenty of questions remain—what will conformance mean, for example, when SCA offers so many options?—yet the technology's potential is evident. The reality today is clear: Anyone who's interested in the future of application development should also be interested in SCA.

## ACKNOWLEDGEMENTS

I'm very grateful to Jim Marino and Michael Rowley at BEA, both of whom patiently answered my many questions about SCA. I'm indebted as well to Eric Newcomer at IONA, who also provided useful assistance.

## ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps information technology professionals around the world understand, use, and make better decisions about enterprise software.