Understanding SCA

(Service Component Architecture)

Jim Marino
Michael Rowley

Independent Technology Guides
David Chappell, Series Editor

Library of Congress Cataloging-in-Publication Data:

Marino, Jim, 1969-
Understanding SCA (Service Component Architecture) / Jim Marino, Michael Rowley.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-51508-7 (pbk. : alk. paper) 1. Application software—Development. 2. Web services. 3. Computer software—Reusability. 4. System design. I. Rowley, Michael. II. Title.
QA76.76.A65M339 2009
005.3—dc22

2009021249

ISBN-13: 978-0-321-51508-7
ISBN-10: 0-321-51508-0
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing July 2009

Editor-in-Chief
Karen Gettman
Executive Editor
Chris Guzikowski
Senior Development Editor
Chris Zahn
Development Editor
Susan Brown Zahn
Managing Editor
Kristy Hart
Project Editor
Jovana San Nicolas-Shirley
Copy Editor
Water Crest Publishing
Indexer
Erika Millen
Proofreaders
Seth Kerney
Apostrophe Editing Services
Publishing Coordinator
Raina Chrobak
Cover Designer
Sandra Schroeder
Compositor
Gloria Schurick

Preface

What is Service Component Architecture (SCA)? What are the key SCA concepts? How will SCA impact technology choices my organization will need to make in the near-term? How should SCA fit into my enterprise architecture? How can I make the best use of SCA in my projects?

Answering these questions is fundamental to understanding SCA. The goal of this book is to help answer those questions by providing the background necessary to use SCA effectively.

Who Can Benefit from This Book

SCA is a technology for creating, assembling, and managing distributed applications. However, this book is not intended solely for developers. Our aim is to benefit "technologists"—developers, but also architects, analysts, managers, and anyone who has a stake implementing information systems—by connecting SCA to broader technology trends.

In this book, we attempt to strike a balance between the "big picture" and the detailed coverage essential to developers. We also endeavor to achieve this balance in a way that is engaging, accurate, and complete.

Both of us have been involved with SCA since its inception, when it started as an informal working group composed of individuals from IBM and BEA (where both of us worked). We were directly involved in shaping SCA as it went through various iterations and changes.

Rather than simply provide a tutorial, we have sought to explain the history and reasoning behind important decisions made during the development of SCA.

Lest we be accused of operating in the "ivory tower" of technology standards, we have also attempted to be informed by practical experience. We have been key contributors to the open source Fabric3 SCA runtime. In addition, while at BEA and now in our current positions, we have had the opportunity to be involved in the development of several large-scale systems built with SCA. We have tried to reflect this experience and lessons learned throughout the book in the form of best practices and implementation advice.

Finally, while we strive for completeness and accuracy, there are inevitably things a book must leave out. SCA is a vast technology that spans multiple programming languages. We have chosen to concentrate on those aspects of SCA that pertain to creating and assembling applications using Java. Although we touch on BPEL, our focus remains on Java, as the latter is a cornerstone of modern enterprise development.

How to Read the Book

Reading a book is like listening to an album (or CD): Both are highly personal experiences. Some prefer to read thoroughly or listen from beginning to end. Others like to skip around, focusing on specific parts.

Understanding SCA is designed to be read in parts but also has a structure tying the various pieces together. The first chapter, "Introducing SCA," provides an overview of SCA and how it fits into today's technology landscape. The second chapter, "Assembling and Deploying a Composite," continues the overview theme by walking through how to build an application using SCA.

Chapter 3, "Service-Based Development Using Java," and Chapter 4, "Conversational Interactions Using Java," respectively, turn to advanced SCA programming model topics. In these chapters, we detail how to design loosely coupled services and asynchronous interactions, manage stateful services, and provide best practices for developing with SCA.

Having explored the SCA programming model in depth, Chapters 5–9 cover the main SCA concepts: composition, policy, wires, bindings, and the domain. In these chapters, we explain how to develop modular applications, use transactions, configure cross-application policies such as security and reliability, integrate with external systems, deploy applications, and structure corporate architectures using SCA.

Chapter 10, "Service-Based Development Using BPEL," demonstrates how to use BPEL with SCA to provide applications with long-running process capabilities.

The final two chapters round out application development with SCA by focusing on the data and presentation tiers. Chapter 11, "Persistence," details how to use Java Persistence API (JPA) with SCA to read and write data from a database. Chapter 12, "The Presentation Tier," demonstrates how to integrate web applications, in particular servlets and JSPs, with SCA services.

2

Assembling and Deploying a Composite

The previous chapter introduced the four core SCA concepts: services, components, composites, and the domain. In this chapter, we explore these in practice by providing a walkthrough of creating a composite and deploying it to a domain. For those wanting to do hands-on development, this chapter also covers using the open source SCA runtime, Fabric3, to deploy and run the composite.

This chapter teaches you the basics of building an SCA application, including the following:

- How to create components that offer services
- How to configure those components and wire them together as part of a composite
- How to expose a service as a web service
- How to package and deploy the composite to a domain

During this exercise, we touch on key SCA design principles and introduce recommended development practices. Subsequent chapters will build on the examples presented here, including designing loosely coupled services, asynchronous communications, and conversational interactions. In these later chapters, we will also cover how to integrate SCA with presentation- and data-tier frameworks.

The `LoanApplication` Composite

Throughout the book, we use a fictitious bank—BigBank Lending—to construct an enterprise-class SCA application. The SCA application we ultimately will build is designed to process loan applications from customers submitted via a web front-end and by independent mortgage brokers via a web service. The high-level application architecture is illustrated in Figure 2.1.

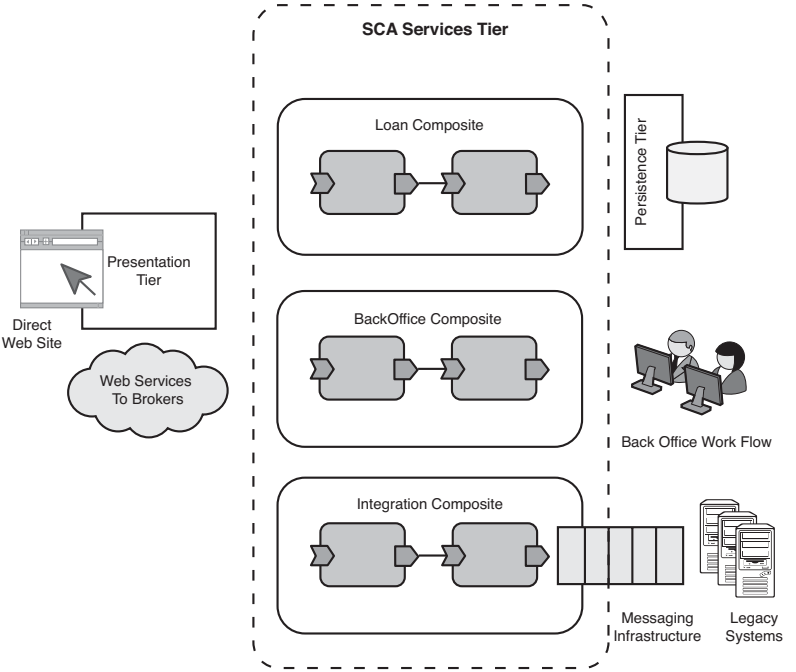

Figure 2.1 The BigBank loan application architecture

The `LoanApplication` composite is the core of BigBank's loan-processing system. It is responsible for receiving loan applications and coordinating with other services to process them. In this chapter, we will start simply by focusing on two Java-based components contained in the composite. `LoanComponent` receives and processes loan application requests from remote clients using web services. It in turn uses the `CreditService` interface implemented by `CreditComponent` to perform a credit check on the applicant (see Figure 2.2).

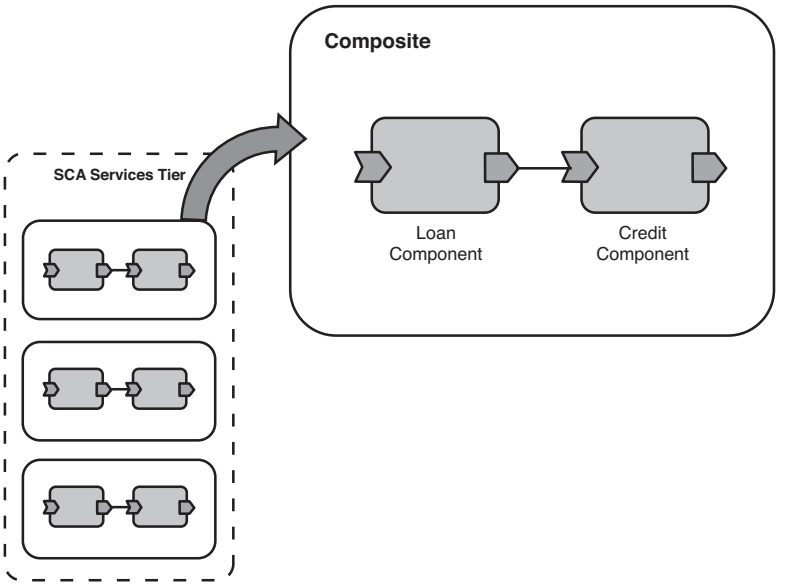

Figure 2.2 The `LoanApplication` composite

The other components—web-front end, data-tier, and integration with external systems—will be covered in later chapters.

Open Source SCA Implementations: Fabric3

Although SCA is an emerging technology, there are already several open source implementations available. Two of the most well known are Fabric3 (http://www.fabric3.org) and Apache Tuscany (http://tuscany.apache.org/). Throughout the book, we use the Fabric3 SCA runtime for hands-on development. Because we (the authors of this book) are involved in the development of Fabric3, you will notice a strong affinity between its capabilities and the topics covered in the book. In addition to support for a majority of the core SCA specifications, Fabric3 provides a number of extensions for popular technologies, including Java Persistence Architecture (JPA) and Hibernate.

Fabric3's design is similar to Eclipse in that it consists of a small core that can be extended through plug-ins. Bindings such as web services, JMS, and RMI are installed as extensions into the Fabric3 core in much the same way that JSP and XML editing

support are added to Eclipse. This gives users the flexibility of choosing just what they need and avoids having to deal with the complexity associated with one-size-fits-all approaches.

This design follows a general trend in software modularity popularized by Eclipse. As development environments increased in complexity in the early 2000s, Eclipse introduced an elegant plug-in mechanism based on OSGi that enabled users to configure their IDE with the specific tools they needed to develop their applications. This greatly reduced software bloat and introduced a new level of flexibility for users. This philosophy has now been extended to runtime architectures as well with the introduction of Profiles in Java EE. Ultimately, modularity benefits users by providing a much more streamlined development, deployment, and management cycle.

Later in the chapter, we provide specific instructions for downloading and getting started with Fabric3. If you want to get a head start, you can download the distribution from http://www.fabric3.org/downloads.html. Be sure to also check out the project mailing lists—they are the best way of getting help should you encounter a problem.

Defining Service Interfaces

Recalling from the previous chapter that components interact through services, we start by defining the service interfaces for the `LoanComponent` and `CreditComponent` components. Because both components are implemented in Java, we use Java to define their service interfaces. The `LoanService` interface is shown in Listing 2.1.

Listing 2.1 *The `LoanService` Interface*

```
@Remotable

public interface LoanService  {

      LoanResult apply(LoanRequest request);

}
```

The `CreditService` interface is presented in Listing 2.2.

Listing 2.2 ***The `CreditService` Interface***

```
@Remotable

public interface CreditService {


      int checkCredit(String id);
}
```

`LoanService` defines one operation, `apply(..)`, which takes a loan application as a parameter. `CreditService` defines one operation, `checkCredit(..)`, which takes a customer ID and returns a numerical credit score. Both interfaces are marked with an SCA annotation, `@Remotable`, which specifies that both services may be invoked by remote clients (as opposed to clients in the same process). Other than the `@Remotable` annotations, the two service contracts adhere to basic Java.

Using Web Services Description Language (WSDL)

In the previous example, we chose Java to define the service contracts for `LoanService` and `CreditService` because it is easy to develop in, particularly when an application is mostly implemented in Java. There are other times, however, when it is more appropriate to use a language-neutral mechanism for defining service contracts. There are a number of interface definition languages, or IDLs, for doing so, but Web Services Description Language (WSDL) is the most accepted for writing new distributed applications. Although labeled as a "web services" technology, WSDL is in fact an XML-based way of describing any service—whether it is exposed to clients as web services—that can be used by most modern programming languages. To understand why WSDL would be used with SCA, we briefly touch on the role it plays in defining service interfaces.

WSDL serves as the lingua franca for code written in one language to invoke code written in another language.

WSDL serves as the lingua franca for code written in one language to invoke code written in another language. It does this by defining a common way to represent operations (what can be invoked), message types (the input and output to operations), and bindings to a protocol or transport (how operations must be invoked). WSDL uses other technologies such as XML Schema to define message

types and SOAP for how invocations are sent over a transport layer (for example, HTTP). Programming languages define mappings to WSDL, making it possible for languages with little in common to communicate, as represented in Figure 2.3.

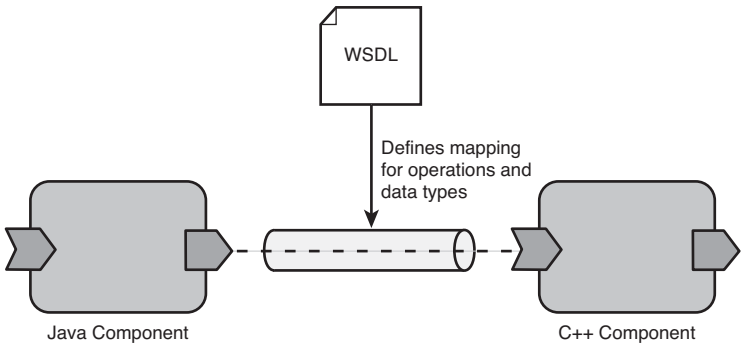

Figure 2.3 WSDL is used to map operations and data types.

Writing WSDL by hand is generally not a pleasant experience; for anything but trivial interfaces, it is a tedious process. Briefly compare the `LoanService` interface previously defined using Java to its WSDL counterpart (see Listing 2.3).

Listing 2.3 ***The `LoanService` WSDL***

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitionsxmlns:ns1="http://loanservice.loanapp/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 name="LoanService" targetNamespace="http://loanser
➥vice.loanapp/">
    <wsdl:message name="applyResponse">
        <wsdl:part element="ns1:applyResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="apply">
        <wsdl:part element="ns1:apply" name=
➥"parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="LoanServicePortType">
        <wsdl:operation name="apply">
            <wsdl:input message="ns1:apply" name="apply">
            </wsdl:input>
```

```
            <wsdl:output message="ns1:applyResponse"
name="applyResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

SCA does not require WSDL to define service interfaces.

Fortunately, SCA does not require WSDL to define service interfaces. Why, then, would someone choose to use WSDL? One scenario where WSDL is used is in top-down development. This style of development entails starting by defining an overall system design, including subsystems and the services they offer, in a way that is independent of the implementation technologies used. WSDL is a natural fit for this approach as it defines service interfaces without specifying how they are to be implemented. In this scenario, an architect could define all service interfaces upfront and provide developers with the WSDLs to implement them.

Defining service contracts using WSDL promotes interoperability.

Few development organizations follow this top-down approach. Typically, service development is iterative. A more practical reason for starting with WSDL is to guarantee interoperability. If a service is created using language-specific means such as a Java interface, even if it is translated into WSDL by tooling, it may not be compatible with a client written in a different language. Using carefully hand-crafted WSDL can reduce this risk.

A third reason to use hand-crafted WSDL is to better accommodate service versioning. Services exposed to remote clients should be designed for loose-coupling. An important characteristic of loose-coupling is that those services should work in a world of mismatched versions where a new version of a service will be backward compatible with old clients. Because WSDL uses XML Schema to define operation parameters, maintaining backward compatibility requires that the parameter-type schemas be designed to handle versioning. This is difficult to do directly in schema but even more difficult using Java classes. In cases where support for versioning is paramount, working directly with WSDL may be the least complex alternative.

One question people typically raise is if SCA does not mandate the use of WSDL, how can it ensure that two components written in

different languages are able to communicate? SCA solves this problem by requiring that all interfaces exposed to remote clients be *translatable* into WSDL. For example, if a service interface is defined using Java, it must be written in such a way that it is possible to represent it in WSDL. This enables a runtime to match a client and service provider by mapping each side to WSDL behind the scenes, saving developers the task of doing this manually.

Given that SCA services available to remote clients must be translatable into WSDL, it is important to note that the latter imposes several restrictions on interface definitions. WSDL stipulates that service interfaces must not make use of operator overloading; in other words, they must not have multiple operations with the same name but different message types. WSDL also requires operation parameters to be expressible using XML Schema. The latter restriction is, in practice, not overly burdensome. Although it might disallow certain data types (for example, Java's `InputStream`), virtually all data types suitable for loosely coupled service interactions can be accommodated by XML Schema. The next chapter will discuss service contract design in detail; for now, it is important to remember these two constraints for services exposed to remote clients.

Services Without WSDL?

Given SCA's heavy reliance on services, it may be surprising that it does not have a canonical interface language. The reasoning behind this decision centers on complexity. Writing WSDL is notoriously difficult. Moreover, previous attempts at defining cross-language IDLs such as CORBA suffered from similar issues. The SCA authors wanted to avoid imposing unnecessary steps in a typical development process. For example, when not doing top-down design, where service interfaces are first defined in a language-neutral format, requiring WSDL is an unnecessary burden, even when tooling can automate some of the process.

When services and service clients are written in the same language, there is no need for a language-neutral representation. In fact, the translation to WSDL can be avoided in some situations where the client and provider are implemented in different languages. For example, languages such as Groovy, BPELJ, and JPython can consume Java interfaces, making WSDL mapping unnecessary. Because distributed applications usually have many components written in the same language, translation into WSDL can usually be avoided.

There are cases where a WSDL-first, top-down design should be used. Sometimes the component implementation technology is not known at the time a system architecture is being designed, or the technology is known but there is a desire to hide it. In those situations, defining interfaces directly in WSDL is appropriate. However, it is a conscious design decision on the part of the SCA authors that a technology should be used only when needed. In the case of WSDL, it is a pragmatic "opt-in" approach to complexity.

Remotable Versus Local Services

For contracts defined using Java, SCA requires that any service exposed across a process boundary be explicitly marked as **remotable**.

Returning to the `LoanService` and `CreditService` interfaces, both are annotated with `@Remotable`, which indicates that a service may, but need not be, accessed remotely. For contracts defined using Java, SCA requires that any service exposed across a process boundary be explicitly marked as **remotable**. Services not marked as remotable—the default case—are **local services**: They are callable only from clients hosted in the same process. In contrast, service interfaces defined by WSDL are remotable by default. This makes sense given that most contracts defined by WSDL are likely to be intended for remote access.

Requiring service contracts to be explicitly marked as remotable indicates which services are designed to be accessible across process boundaries. The distinction is necessary because local and remotable services have different behavior. The next chapter covers these differences at length, which we briefly describe here.

Remotable Services Must Account for Network Latency

Remotable services should be coarse-grained.

Clients of remotable services must accommodate network latency. This means that remotable services should be coarse-grained—that is, they should contain few operations that are passed larger data sets, as opposed to a number of individual operations that take a small number of parameters. This reduces the degree of network traffic and latency experienced by clients. In addition, remotable services often define asynchronous operations as a way to handle network latency and service interruptions. Local services are not subject to these demands as calls occur in the same process. Therefore, they tend to be finer-grained and use synchronous operations.

Clients of Remotable Services May Experience Communications Failures

Because invocations on remotable services generally travel over a network, there is a possibility communications may be interrupted. In SCA, the unchecked `org.osoa.sca.ServiceUnavailableException` exception will be thrown if a communication error occurs. Clients need to handle such exceptions, potentially by retrying or reporting an error.

Remotable Services Parameters Are Passed by Value

Parameters associated with remotable service operations behave differently than those of operations on local services.

Parameters associated with remotable service operations behave differently than those of operations on local services. When remotable invocations are made, parameters are marshaled to a protocol format such as XML and passed over a network connection. This results in a copy of the parameters being made as the invocation is received by the service provider. Consequently, modifications made by the service provider will not be seen by the client. This behavior is termed "pass-by-value." In contrast, because invocations on local services are made in the same process, operation parameters are not copied. Any changes made by the service provider will be visible to the client. This behavior is known as "pass-by-reference." Marking a service as remotable signals to clients whether pass-by-value or pass-by-reference semantics will be in effect.

Table 2.1 summarizes the differences between remotable and local services.

Table 2.1 Remotable Versus Local Services

Remotable Services	Local Services
Are invoked in-process and remotely.	Are always invoked in-process.
Parameters are pass-by-value.	Parameters are pass-by-reference.
Are coarse-grained.	Tend to be fine-grained.
Are loosely coupled and favor asynchronous operations.	Commonly use synchronous operations.

Local Services and Distributed Systems

It may seem odd that a technology designed for building distributed applications specifies local service contracts as the default when defined in Java. This was a conscious decision on the part of the SCA authors. Echoing Jim Waldo's seminal essay, "The Fallacies of Distributed Computing," location transparency is a fallacy: Crossing remote boundaries requires careful architectural consideration that has a direct impact on application code. Issues such as network latency, service availability, and loose coupling need to be accounted for in component implementations. This was one of the lessons learned with EJB: Many early Java EE applications suffered from crippling performance bottlenecks associated with making too many remote calls.

To minimize remote calls, distributed applications have a relatively small number of services exposed to remote clients. Each of these services should in turn have a few coarse-grained operations that perform a significant task, such as processing a loan application or performing an inventory check. Moreover, these services should be carefully constructed so that new versions can be deployed without breaking existing clients. Limiting the number of remotable services and operations helps avoid performance issues and facilitates versioning by restricting change to a few areas in an application.

Given the lessons learned from previous distributed system technologies, the designers of SCA were faced with a dilemma: how to support applications built using coarse-grained services that did not repeat the problems of the past. The answer was, ironically, to provide good support for *fine-grained,* local services. If the only way to get the benefits of SCA such as programming model simplicity were to use remotable services, developers would be pushed into making all code remotable, even if it should not be. By providing a model for local services, remote boundaries can be chosen carefully, exposing only those parts of an application that should be accessible to clients hosted in different processes.

Creating Component Implementations

Well-designed service-based architectures typically have a limited number of coarse-grained services that coordinate other services to perform specific tasks. The heart of the `LoanApplication` composite is `LoanComponent`, which is responsible for receiving loan application data through its `LoanService` interface and delegating to other services for processing. The implementation is a basic Java class that takes a reference proxy to a `CreditService` interface as

Well-designed service-based architectures typically have a limited number of coarse-grained services that coordinate other services to perform specific tasks.

part of its constructor signature. The `LoanComponent` component uses the service to provide a credit score for the applicant. When reviewing the implementation, take note of the `@Reference` annotation in the constructor (see Listing 2.4).

Listing 2.4 *The `LoanComponent` Implementation*

```
public class LoanComponent  implements LoanService {
      private CreditService service;



      public void LoanComponent   (@Reference CreditService service){
            this.service = service;
      }

      public LoanResult apply(LoanRequest request) {
            // ....
      }

      public int checkStatus(String applicationID){
            // ....
      }
}
```

In Listing 2.4, the `@Reference` annotation instructs the SCA runtime that `LoanComponent` requires a reference to `CreditService`. An implementation of `CreditService` is provided by `CreditComponent`, shown in Listing 2.5.

Listing 2.5 *The `CreditComponent` Implementation*

```
public class CreditComponent implements CreditService {

      public int checkCredit(String id){
            // ....
      }
}
```

Although the code has been simplified from what would be typically encountered in a real-world scenario, the implementation—like `LoanComponent`—is straight Java. Even though both components may be hosted on different machines, the only thing required to facilitate remote communication is the presence of `@Remotable` on the `CreditService` interface.

A Note on OASIS and OSOA Java APIs and Annotations

As mentioned previously, prior to moving to OASIS, SCA was part of the Open SOA (OSOA) collaboration effort. While at OSOA, the Java APIs and annotations used throughout this book are published under the org.*osoa.sca* package. As part of the move to OASIS, the Java APIs and annotations will also be published under the *org.oasisopen.sca* package. We have decided to continue to use the OSOA package version because, at the time of this writing, the OSOA annotations are more prevalent.

SCA leaves the heavy lifting associated with establishing remote communications to the runtime, as opposed to application code and API calls.

SCA leaves the heavy lifting associated with establishing remote communications to the runtime, as opposed to application code and API calls. As we saw in the introductory chapter, SCA does this through **wires**. Conceptually, a wire is a connection provided by the runtime to another service. A wire is specified—in this case, the wire between `LoanComponent` and `CreditComponent`—in the composite file, which we show in the next section. For now, we will assume a wire has been specified and describe how an SCA runtime goes about connecting `LoanComponent` to the `CreditService` interface of `CreditComponent`.

In Java, the runtime provides a wire by doing one of the following: calling a setter method annotated with `@Reference` and passing in a reference to the service; setting a field marked with `@Reference`; or passing a reference to the service as a constructor parameter annotated with `@Reference`, as in the example given previously in Figure 2.3.

In actuality, when the SCA runtime injects the `CreditService`, it is likely not a "direct" reference to `CreditComponent` but instead a generated "proxy" that implements the `CreditService` interface (see Figure 2.4).

The proxy is responsible for taking an invocation and flowing it to the target service, whether it is co-located or hosted in a remote JVM. From the perspective of `LoanComponent`, however, `CreditService` behaves as a typical Java reference.

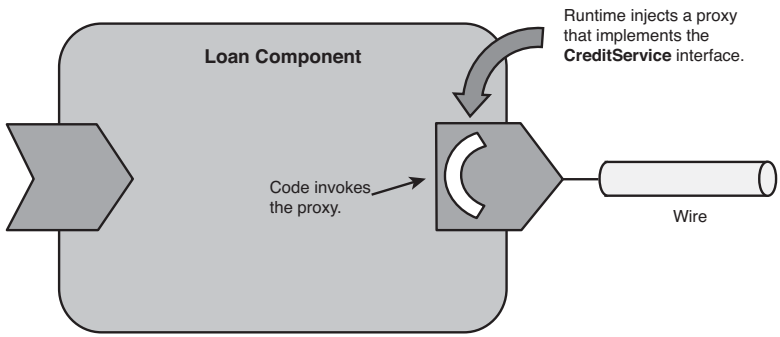

Figure 2.4 Reference proxy injection

SCA is based on Inversion of Control (IoC), also known as dependency injection.

An important characteristic of wires is that their details are hidden from the client implementation. In our example, `LoanComponent` does not have knowledge of the wire communication protocol or the address of `CreditService`. This approach will be familiar to Spring developers. SCA is based on Inversion of Control (IoC), also known as dependency injection, popularized by the Spring framework. Instead of requiring a component to find its dependent services through a service locator API and invoke them using transport-specific APIs, the runtime provides service references when an instance is created. In this case, `CreditService` is injected as a constructor parameter when `LoanComponent` is instantiated.

There are a number of advantages to IoC. Because the endpoint address of `CreditService` is not present in application code, it is possible for a system administrator or runtime to make the decision at deployment whether to co-locate the components (possibly for performance reasons) or host them in separate processes. Further, it is possible to "rewire" `LoanComponent` to another implementation of `CreditService` without having to change the `LoanComponent` code itself. And, because the client does not make use of any protocol-specific APIs, the actual selection of a communication protocol can be deferred until deployment or changed at a later time.

Injection Styles

In the current version of `LoanComponent`, we elected to define the reference to `CreditService` as a constructor parameter. This is commonly referred to as constructor-based injection. Some developers prefer to inject dependencies through setter methods or

directly on fields. The SCA Java programming model accommodates these alternative approaches as well by supporting injecting references on methods and fields. We will take a closer look at each injection style in turn.

Constructor-Based Injection

Constructor-based injection has the advantage of making dependencies explicit at compile time.

The primary drawback of constructor-based injection is that the constructor parameter list can become unwieldy for components that depend on a number of services.

Constructor-based injection has the advantage of making dependencies explicit at compile time. In our example, `LoanComponent` cannot be instantiated without `CreditService`. This is particularly useful for testing, where component implementations are instantiated directly in test cases. Constructor-based injection also enables fields to be marked as final so that they cannot be inadvertently changed later on. When other forms of injection are used, final fields can't be used. The primary drawback of constructor-based injection is that the constructor parameter list can become unwieldy for components that depend on a number of services.

In some cases, component implementations may have more than one constructor. The SCA Java programming model defines a rule for selecting the appropriate constructor in cases where there is more than one. If one constructor has parameters marked with `@Reference` or `@Property`, it will be used. Otherwise, a developer can explicitly mark a constructor with the SCA `@Constructor` annotation, as shown in Listing 2.6.

Listing 2.6 ***The `@Constructor` Annotation***

```
@Constructor
public CreditComponent  (double min, double max) {
   // …
}
```

Setter-Based Injection

SCA supports method-based reference injection as an alternative to constructor-based injection.

SCA supports method-based reference injection as an alternative to constructor-based injection. For example, `LoanComponent` could have been written as shown in Listing 2.7.

Listing 2.7 ***Setter-Based Injection***

```
public class LoanComponent{

      public LoanComponent () {}
```

```
        @Reference
        public setCreditService(CreditService creditService) {
              // …
        }
}
```

When `LoanComponent` is instantiated, the SCA runtime will invoke the `setCreditService` method, passing a reference proxy to `CreditService`. An important restriction SCA places on this style of injection is that setter methods must be either public or protected; private setter methods are not allowed because it violates the object-oriented principle of encapsulation. (That is, private methods and fields should not be visible outside a class.)

The main benefit of setter-based injection is that it allows for reinjection of wires dynamically at runtime.

The main benefit of setter-based injection is that it allows for reinjection of wires dynamically at runtime. We cover wire reinjection in Chapter 7, "Wires."

There are two major downsides to setter injection. Component dependencies are dispersed across a number of setter methods, making them less obvious and increasing the verbosity of the code because a method needs to be created for every reference. In addition, setter methods make references that potentially should be immutable subject to change, because the fields they are assigned to cannot be declared final.

Setter Injection Best Practices

There are a couple of best practices to keep in mind when using setter-based injection. First, setter methods should not be part of the service interface because they are implementation details. For example, `LoanService` does not define the method `setCreditService(CreditService creditService)`—the fact that `LoanComponent` uses `CreditService` is an implementation detail clients should not be aware of.

Second, avoid making setter methods protected, even though SCA allows this. Doing so makes unit testing difficult because unit tests would need to either subclass the component implementation to override the setters and make them public or use reflection to set them directly. If setter methods are not part of a service contract, there is no risk a client will inadvertently invoke them if they are made public.

Field-Based Injection

The final form of injection supported by SCA is field-based. This style enables fields to be directly injected with reference proxies (see Listing 2.8).

Listing 2.8 ***Field-Based Injection***

```
public class LoanComponent   {

      @Reference
      protected CreditService CreditService;

      //....
}
```

Field-injection follows the basic pattern set by method-injection except that they may be private and public or protected. In the absence of a name attribute declared on `@Reference`, the field name is used as the name of the reference. Again, the preceding example would be configured using the same composite syntax as the previous examples.

A major advantage of field-based injection is that it is concise. (Methods do not need to be created for each reference.) It also avoids long constructor parameter lists. The main disadvantage of field-based injection is it is difficult to unit test; component classes must either be subclassed to expose reference fields or those fields must be set through Java reflection.

A major advantage of field-based injection is that it is concise.

The main disadvantage of field-based injection is it is difficult to unit test.

Perspective: What's the Best Injection Style?

Several years ago, setter- versus constructor-based injection was an area of contention among advocates of various Java-based IoC frameworks, notably Spring and PicoContainer. Most modern IoC frameworks now support both approaches, as does SCA.

In the process of writing this book, we debated between ourselves about the best injection style. Jim favors constructor injection because it makes service dependencies explicit. Mike prefers field-based injection because it limits verbosity. In the end, like the debates among the various IoC frameworks a few years back, we went

around in circles and were unable to convince one another. This led us to agree on an important point: Choosing an injection style is largely a matter of personal preference. Pick the one that best suits the project requirements or the one project developers are used to and stay consistent.

That said, there is one important difference between field and setter versus constructor injection in SCA. Namely, field and setter injection can be dynamic. As we will cover in Chapter 7, field- and setter-based references may be reinjected if a reference changes after a component has been instantiated. In contrast, constructor-based references cannot be changed. If a reference may change, you need to use field- or setter-based injection. On the other hand, if a reference must be immutable, use constructor injection.

Defining Properties

Consider the case where we want to add the capability to set configuration parameters on the `CreditComponent` component, such as minimum and maximum scores. SCA supports configuration through component **properties**, which in Java are specified using the `@Property` annotation. `CreditComponent` is modified to take maximum and minimum scores in Listing 2.9.

Listing 2.9 ***Defining Component Properties***

```
public void CreditComponent implements CreditService {

      private int min;
      private int max;

      public CreditComponent (@Property(name="min") int min,
                              @Property(name="max") int max) {
            this.min = min;
            this.max = max;
      }
      // ....
}
```

Like a reference, a property name is specified using the "name" attribute, whereas the "required" attribute determines whether a property value must be provided in the composite file (that is, when it is set to true) or it is optional (that is, it is set to false, the default).

In addition, properties follow the same injection guidelines as references: constructor-, method-, and field-based injection are supported.

References provide access to services, whereas properties provide configuration data.

Given that most IoC frameworks do not distinguish between properties and references, why does SCA? The short answer is they are different. References provide access to services, whereas properties provide configuration data. Differentiating properties and references makes it clear to someone configuring a component whether a property value needs to be supplied or a reference wired to a service. Further, as we will see in later chapters, references may have various qualities of service applied to them, such as reliability, transactions, and security. The benefits of distinguishing properties and references also extends to tooling: Knowing if a particular value is a property or reference makes for better validation and visual feedback, such as displaying specific icons in graphical tooling.

Assembling the `LoanApplication` Composite

Listing 2.10 provides a complete version of the `LoanApplication` composite we first introduced in the last chapter. Let's examine it in the context of the `LoanComponent` and `CreditComponent` implementations we have just discussed.

Listing 2.10 ***The `LoanApplication` Composite***

```
<composite xmlns=http://www.osoa.org/xmlns/sca/1.0

      targetNamespace="
http://www.bigbank.com/xmlns/loanApplication/1.0"

      name="LoanApplication">

   <component name ="LoanComponent">
      <implementation.java class="com.acme.LoanComponent  "/>
      <property name="currency">USD</property>
      <reference name="creditService" target="CreditComponent "/>
   <component>

   <component name = "CreditComponent">
      <implementation.java class="com.acme.CreditComponent "/>
   <component>
</composite>
```

Composites include `targetNamespace` *and* `name` *attributes, which together form their* ***qualified name****, or QName.*

Composites include `targetNamespace` and `name` attributes, which together form their **qualified name**, or QName. The QName of the `LoanApplication` composite is http://www.bigbank.com/xmlns/loanApplication/1.0:LoanApplication. QNames are similar to the combination of package and class name in Java: They serve to uniquely identify an XML artifact—in this case, a composite. The `targetNamespace` portion of the QName can be used for versioning. In the example, the `targetNamespace` ends with 1.0, indicating the composite version. The version should be changed any time a nonbackward-compatible change is made to the definition (and should not be changed otherwise).

Continuing with the composite listing in Listing 2.10, `LoanComponent` and `CreditComponent` are defined by the `<component>` element. Both component definitions contain an entry, `<implementation.java>`, which identifies the Java class for the respective component implementations. If the components were implemented in BPEL, the `<implementation.bpel>` element would have been used, as follows:

```
<implementation.bpel process="bb:LoanApplicationProcess">
```

The `<reference>` element in the `LoanComponent` definition configures the reference to `CreditService`, as follows:

```
<reference name="creditService" target="CreditService"/>
```

Recalling that the `LoanComponent` implementation declares a reference requiring a `CreditService` in its constructor, we get the following:

```
public LoanComponent  (@Reference (name="CreditService") CreditService
➥creditService) {
// …
}
```

The `<reference>` element configures the `creditService` reference by wiring it to the `CreditService` provided by `CreditComponent`. When an instance of `LoanComponent` is created by the SCA runtime, it will pass a proxy to `CreditService` as part of the constructor invocation.

Properties are configured in a composite file using the `<property>` element. In the `LoanApplication` composite, `CreditComponent` is configured with `min` and `max` values (see Listing 2.11).

Listing 2.11 ***Configuring Property Values***

```
<component name="CreditComponent">
      <implementation.java class=".."/>
      <property name="min">300</property>
      <property name="max">850</property>
</component>
```

The property values will be injected into the component by the runtime when a component instance is created.

It is important to note the naming convention used for configuring references and properties defined on setter methods. In the absence of an explicit name attribute on `@Reference` or `@Property` annotation, the name of the reference is inferred from the method name according to JavaBean semantics. In other words, for method names of the form "`setXXXX`," the `set` prefix is dropped and the initial letter of the remaining part is made lowercase. Otherwise, the value specified in the name attribute is used.

An interesting characteristic of reference and property configuration in a composite is that the format remains the same, regardless of the style of injection used in the implementation. For example, the following component entry

```
<component name="LoanComponent">
      <implementation.java class=".."/>
      <reference name="creditScoreService" target="CreditComponent "/>
</component>
```

configures a reference specified on a constructor parameter,

```
public LoanComponent  (@Reference(name="creditScoreService"
➥CreditService CreditService) {
      // …
}
```

or a setter method,

```
@Reference

public void setCreditScoreService(CreditService creditScoreService){

      //…
}
```

or a field:

```
@Reference

protected CreditService creditScoreService;
```

Binding a Web Service Endpoint

Bindings are used to specify the communication protocol a service is available over, such as web services, RMI, or plain XML over HTTP.

The `LoanApplication` composite would be more useful if its services were made accessible to clients that are outside the SCA domain—for example, to independent mortgage broker systems. In SCA, services are exposed to external clients over a **binding**. Bindings are used to specify the communication protocol over which a service is available, such as web services, RMI, or plain XML over HTTP (without the SOAP envelope). A service may be exposed over more than one binding, providing multiple ways for external clients to invoke it. For example, the `LoanService` could be bound to web services and a proprietary EDI protocol (see Figure 2.5).

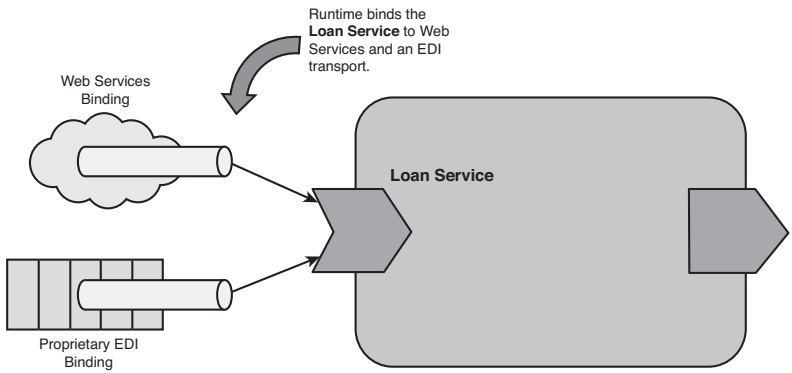

Figure 2.5 Binding the `LoanService`

Bindings can be added or removed in runtimes that support dynamic updates.

Moreover, bindings can be added or removed in runtimes that support dynamic updates. For example, after clients have transitioned to using web services, the EDI binding for the `LoanService` interface could be deprecated and eventually phased out. Alternatively, a high-speed binary binding could be added for clients requiring improved performance (such as a binding based on the new W3C Efficient XML for Interchange format, EXI).

Service bindings are specified in the composite file using a combination of service and binding elements. Listing 2.12 binds the `LoanService` interface to web services.

Listing 2.12 ***Binding the LoanService Interface as a Web Service Endpoint***

```
<service name="LoanService">
      <binding.ws>
</service>
```

When LoanComponent is activated in the domain, the SCA infrastructure is responsible for making LoanService available as a web service.

The exact mechanics of how this binding is achieved are runtime-dependent. However, all SCA implementations must perform the following steps (which will generally be transparent to the person deploying a composite). First, if no WSDL is specified, the runtime will need to generate it based on the LoanService Java interface. This will entail creating a WSDL document similar to the one listed at the beginning of the chapter, but also including WSDL binding and WSDL service elements. (The algorithm for generating the WSDL is standardized by SCA.) After the WSDL is generated, the runtime will need to make the service and WSDL available to clients as a web service at the endpoint address listed in the WSDL. Depending on the runtime, this may involve deploying or dynamically configuring middleware such as creating a HTTP listener for the service on a particular machine. Fortunately, SCA hides the complexities of this process, so people deploying composites need not worry about how this is actually done.

Packaging the LoanApplication Composite

SCA specifies one interoperable packaging format for composite files and associated artifacts such as Java classes, XSDs, and WSDLs: the ZIP archive.

SCA specifies one interoperable packaging format for composite files and associated artifacts such as Java classes, XSDs, and WSDLs: the ZIP archive. However, to accommodate the diverse range of packaging formats used by various programming languages, SCA allows runtimes to support other formats in addition to the ZIP archive. A C++ runtime may accept DLLs; a runtime may also support various specializations of the ZIP format. Fabric3 also supports JARs and Web Archives (WARs).

The sca-contribution.xml file provides SCA-specific information about the contents of the archive.

SCA ZIP archives include a metadata file, sca-contribution.xml, in the META-INF directory. The sca-contribution.xml file provides SCA-specific information about the contents of the archive, most notably the composites available for deployment. In general, one

deployable composite will be packaged in an archive, although in some cases (which we discuss in later chapters), no deployable composites or multiple deployable composites may be present.

A contribution is an application artifact that is "contributed" or made available to a domain.

The name *sca-contribution.xml* derives from SCA terminology: A contribution is an application artifact that is "contributed" or made available to a domain. A contribution can be a complete composite and all the artifacts necessary to execute it, or it might just contain artifacts to be used by composites from other contributions, such as a library, XSDs, or WSDLs. `LoanApplication` is packaged as a complete composite. Its sca-contribution.xml is shown in Listing 2.13.

Listing 2.13 ***A Contribution Manifest***

```
<contribution xmlns=http://www.osoa.org/xmlns/sca/1.0

xmlns:bb="http://www.bigbank.com/xmlns/lending/composites/1.0">

    <deployable composite="bb:LoanApplication"/>

</contribution>
```

The `<deployable>` element identifies a composite available for deployment contained in the archive. In this case, it points to the name of the `LoanApplication` composite, as defined in the `<composite>` element of its .composite file:

```
<composite  xmlns="http://www.osoa.org/xmlns/sca/1.0"

targetNamespace="http://www.bigbank.com/xmlns/lending/composites/1.0"
              name="LoanApplication"…>
```

SCA does not specify a location for composite files; they can be included in any archive directory.

Unlike sca-contribution.xml, SCA does not specify a location for composite files; they can be included in any archive directory. However, as a best practice, it is recommended that deployable composite files be placed alongside sca-contribution.xml in the `META-INF` directory so they can be easily found.

Deploying the `LoanApplication` Composite

Composites can be deployed to a domain using a variety of mechanisms. In a test environment, deployment may involve placing the contribution archive in a file system directory. In production environments, where tighter controls are required, deployment would typically be performed through a command-line tool or script.

Conceptually, deployment involves contributing a composite to a domain and activating its components, as depicted in Figure 2.6.

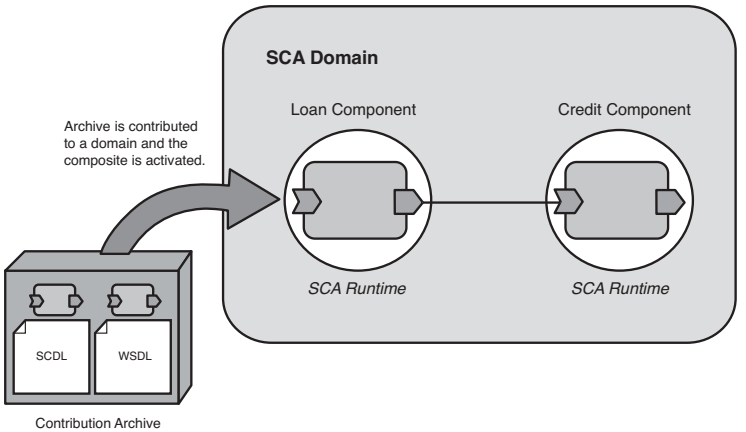

Figure 2.6 Deploying and activating the `LoanApplication` composite

When the `LoanApplication` composite is deployed, the SCA runtime instantiates `LoanComponent` and `CreditComponent`. During this process, because `LoanService` is configured with the web services binding, it is exposed as a web service endpoint. When the `LoanApplication` composite is activated in the domain, its components are available to process client requests.

Using Fabric3

Having completed the walkthrough of assembling and packaging the `LoanApplication` composite, we put this knowledge to practice by deploying a sample application to the Fabric3 SCA runtime.

Fabric3 has a modular architecture similar to Eclipse.

Fabric3 is a full-featured, open source SCA implementation. It has a highly modular design with preconfigured distributions for a number of environments. For example, Fabric3 has distributions that can be embedded in a servlet container, such as Tomcat or Jetty, and specific Java EE application servers, including JBoss, WebLogic, and WebSphere.

Fabric3 has a modular architecture similar to Eclipse. The core distributions implement basic SCA functionality, whereas additional features are added through extensions. This allows Fabric3 to remain lightweight and allows users to include only the features required by their applications. For example, support for bindings such as web services is added as extensions to the core.

To get started with deploying the loan application, you will need to set up Fabric3 and your development environment. We assume that you have JDK 5.0 installed on your machine. To configure your machine, perform the steps outlined in the following sections.

Download Fabric3 `LoanApplication` Sample

Fabric3 provides a `LoanApplication` sample that we use in this hands-on exercise. The sample is a full-fledged version of the loan-processing system covered in this chapter and includes integration with JPA and a web application front-end. It can be downloaded from the same place the Fabric3 distribution is located: http://www.fabric3.org/downloads.html.

The sample contains a utility for downloading the Fabric3 runtime and extensions. Follow the instructions to run the utility and install the runtime.

Verify the Installation

To verify that the server has been successfully installed, go to the `bin` directory where it has been installed and execute `java –jar server.jar`. This will start the server.

Build and Deploy the Application

We are now ready to build and deploy the application. First, follow the instructions for building the sample application. After this is done, start the Fabric3 server by issuing the following command from the `bin` directory where it is installed:

```
java –jar server.jar
```

When the server starts, it activates an SCA domain that is contained in a single process. In a distributed environment, multiple Fabric3 servers participate in a single domain that spans processes.

After the server has booted, copy the loan application JAR that was built in the previous step from the target directory to the deploy directory of the Fabric3 server installation. The server will then deploy the application to the domain.

Invoking the `LoanApplication` Service

After the application has been deployed, we can invoke the `LoanService` interface. The sample application contains a JAX-WS client that can be used to test-drive the service. Follow the instructions for launching the test-client from the command line.

This completes the hands-on walkthrough of building and deploying an SCA application with Fabric3. At this point, it is worth spending some time familiarizing yourself with the application code. As you will see, most of the tedious tasks of generating WSDLs and exposing web services are handled transparently by the runtime. In the following chapters, we expand the loan application by introducing additional SCA features and capabilities. However, the basic structure and simplicity of the code will remain the same.

Summary

We have covered significant ground in this chapter, providing a detailed discussion of key SCA concepts and design principles. Specifically, we have accomplished the following:

- Defined service contracts
- Written component implementations using the SCA Java programming model
- Configured components as part of a composite
- Exposed an SCA service using web services
- Deployed a composite to an SCA runtime

With this foundation in place, we turn our attention in the next chapter to designing and building loosely coupled services using Java.

Index

D

L

M

Q

R

S

T

U-V

W

X-Y-Z