Declared XML Namespace(s):
http://docs.oasis-open.org/ns/opencsa/sca/200712

Abstract:
The SCA Java Common Annotation and APIs specify a Java syntax for programming concepts defined in the SCA Assembly Model Specification. It specifies a set of APIs and annotations that can be used by Java-based SCA specifications.
Specifically, this specification covers:
1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations
Note that individual programming models can chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

Status:
This document was last revised or approved by the OASIS Service Component Architecture / J (SCA-J) TC on the above date. The level of approval is also listed above. Check the “Latest Version” or “Latest Approved Version” location noted above for possible later revisions of this document.
Technical Committee members should send comments on this specification to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the “Send A Comment” button on the Technical Committee’s web page at http://www.oasis-open.org/committees/sca-j/.
For information on whether any patents have been disclosed that might be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/sca-j/ipr.php).
The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/sca-j/.
Notices

Copyright © OASIS® 2005, 2008. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The names "OASIS", [insert specific trademarked names and abbreviations here] are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/who/trademark.php for above guidance.
# Table of Contents

1. **Introduction** ........................................................................................................ 6
   1.1 **Terminology** ........................................................................................................ 6
   1.2 **Normative References** .......................................................................................... 6
   1.3 **Non-Normative References** ................................................................................... 7
2. **Implementation Metadata** ....................................................................................... 8
   2.1 **Service Metadata** ............................................................................................... 8
      2.1.1 **@Service** ........................................................................................................ 8
      2.1.2 **Java Semantics of a Remotable Service** ............................................................ 8
      2.1.3 **Java Semantics of a Local Service** ................................................................... 8
      2.1.4 **@Reference** .................................................................................................... 9
      2.1.5 **@Property** ....................................................................................................... 9
      2.2 **Implementation Scopes: @Scope, @Init, @Destroy** .............................................. 9
      2.2.1 **Stateless scope** ............................................................................................. 9
      2.2.2 **Composite scope** ......................................................................................... 10
3. **Interface** .................................................................................................................. 11
   3.1 **Java interface element – <interface.java>** ............................................................ 11
   3.2 **@Remotable** ....................................................................................................... 12
   3.3 **@Callback** .......................................................................................................... 12
4. **Client API** ............................................................................................................... 13
   4.1 **Accessing Services from an SCA Component** ......................................................... 13
      4.1.1 **Using the Component Context API** ................................................................. 13
   4.2 **Accessing Services from non-SCA component implementations** ........................... 13
      4.2.1 **ComponentContext** ..................................................................................... 13
5. **Error Handling** ....................................................................................................... 14
6. **Asynchronous Programming** .................................................................................. 15
   6.1 **@OneWay** ............................................................................................................ 15
   6.2 **Callbacks** .......................................................................................................... 15
      6.2.1 **Using Callbacks** ........................................................................................... 15
      6.2.2 **Callback Instance Management** .................................................................... 17
      6.2.3 **Implementing Multiple Bidirectional Interfaces** ............................................. 17
      6.2.4 **Accessing Callbacks** .................................................................................... 18
7. **Policy Annotations for Java** .................................................................................... 19
   7.1 **General Intent Annotations** .................................................................................. 19
   7.2 **Specific Intent Annotations** .................................................................................. 21
      7.2.1 **How to Create Specific Intent Annotations** ..................................................... 21
   7.3 **Application of Intent Annotations** ....................................................................... 22
      7.3.1 **Inheritance And Annotation** .......................................................................... 22
   7.4 **Relationship of Declarative And Annotated Intents** ............................................. 24
   7.5 **Policy Set Annotations** ....................................................................................... 24
   7.6 **Security Policy Annotations** ................................................................................ 25
      7.6.1 **Security Interaction Policy** ............................................................................. 25
      7.6.2 **Security Implementation Policy** ..................................................................... 26
8. **Java API** .................................................................................................................. 29

---

23 February 2009

Copyright © OASIS® 2005, 2009. All Rights Reserved.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Component Context</td>
<td>29</td>
</tr>
<tr>
<td>8.2</td>
<td>Request Context</td>
<td>30</td>
</tr>
<tr>
<td>8.3</td>
<td>ServiceReference</td>
<td>31</td>
</tr>
<tr>
<td>8.4</td>
<td>ServiceRuntimeException</td>
<td>31</td>
</tr>
<tr>
<td>8.5</td>
<td>ServiceUnavailableException</td>
<td>32</td>
</tr>
<tr>
<td>8.6</td>
<td>InvalidServiceException</td>
<td>32</td>
</tr>
<tr>
<td>8.7</td>
<td>Constants Interface</td>
<td>32</td>
</tr>
<tr>
<td>9.1</td>
<td>@AllowsPassByReference</td>
<td>33</td>
</tr>
<tr>
<td>9.2</td>
<td>@Authentication</td>
<td>33</td>
</tr>
<tr>
<td>9.3</td>
<td>@Callback</td>
<td>34</td>
</tr>
<tr>
<td>9.4</td>
<td>@ComponentName</td>
<td>35</td>
</tr>
<tr>
<td>9.5</td>
<td>@Confidentiality</td>
<td>36</td>
</tr>
<tr>
<td>9.6</td>
<td>@Constructor</td>
<td>37</td>
</tr>
<tr>
<td>9.7</td>
<td>@Context</td>
<td>37</td>
</tr>
<tr>
<td>9.8</td>
<td>@Destroy</td>
<td>38</td>
</tr>
<tr>
<td>9.9</td>
<td>@EagerInit</td>
<td>38</td>
</tr>
<tr>
<td>9.10</td>
<td>@Init</td>
<td>39</td>
</tr>
<tr>
<td>9.11</td>
<td>@Integrity</td>
<td>39</td>
</tr>
<tr>
<td>9.12</td>
<td>@Intent</td>
<td>40</td>
</tr>
<tr>
<td>9.13</td>
<td>@OneWay</td>
<td>41</td>
</tr>
<tr>
<td>9.14</td>
<td>@PolicySet</td>
<td>41</td>
</tr>
<tr>
<td>9.15</td>
<td>@Property</td>
<td>42</td>
</tr>
<tr>
<td>9.16</td>
<td>@Qualifier</td>
<td>43</td>
</tr>
<tr>
<td>9.17</td>
<td>@Reference</td>
<td>44</td>
</tr>
<tr>
<td>9.17.1</td>
<td>Reinjection</td>
<td>46</td>
</tr>
<tr>
<td>9.18</td>
<td>@Remotable</td>
<td>48</td>
</tr>
<tr>
<td>9.19</td>
<td>@Requires</td>
<td>49</td>
</tr>
<tr>
<td>9.20</td>
<td>@Scope</td>
<td>50</td>
</tr>
<tr>
<td>9.21</td>
<td>@Service</td>
<td>51</td>
</tr>
<tr>
<td>10</td>
<td>WSDL to Java and Java to WSDL</td>
<td>53</td>
</tr>
<tr>
<td>10.1</td>
<td>JAX-WS Client Asynchronous API for a Synchronous Service</td>
<td>53</td>
</tr>
<tr>
<td>A</td>
<td>XML Schema: sca-interface-java.xsd</td>
<td>55</td>
</tr>
<tr>
<td>B</td>
<td>Conformance Items</td>
<td>56</td>
</tr>
<tr>
<td>C</td>
<td>Acknowledgements</td>
<td>62</td>
</tr>
<tr>
<td>D</td>
<td>Non-Normative Text</td>
<td>63</td>
</tr>
<tr>
<td>E</td>
<td>Revision History</td>
<td>64</td>
</tr>
</tbody>
</table>
1 Introduction

The SCA Common Annotation, APIs, Client and Implementation Model specifies a Java syntax for programming concepts defined in the SCA Assembly Model Specification [ASSEMBLY]. It specifies a set of APIs and annotations that can be used by Java-based SCA specifications.

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous services
4. Metadata for callbacks
5. Definitions of standard component implementation scopes
6. Java to WSDL and WSDL to Java mappings
7. Security policy annotations

Note that individual programming models can choose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

The goal of specifying the annotations, APIs, client and implementation model in this specification is to promote consistency and reduce duplication across various Java-related component implementation type specifications. The annotations, APIs, and client and implementation model defined in this specification are designed to be used by other SCA Java-related specifications in either a partial or complete fashion.

This document defines implementation metadata using the annotation capability from Java™ 2 Standard Edition (J2SE) 5. However, SCA also allows service clients and implementations to be written using J2SE 1.4. All metadata that is represented by annotations can also be expressed using a component type side file, as defined in the SCA Assembly Specification [ASSEMBLY].

1.1 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

1.2 Normative References

[WSDL] WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsd1, WSDL 2.0: http://www.w3.org/TR/wsd20/
1.3 Non-Normative References

[EBNF-Syntax]  Extended BNF syntax format used for formal grammar of constructs
http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation
2 Implementation Metadata

This section describes SCA Java-based metadata, which applies to Java-based implementation types.

2.1 Service Metadata

2.1.1 @Service

The @Service annotation is used on a Java class to specify the interfaces of the services implemented by the implementation. Service interfaces are defined in one of the following ways:

- As a Java interface
- As a Java class
- As a Java interface generated from a Web Services Description Language [WSDL] (WSDL) portType (Java interfaces generated from a WSDL portType are always remotable)

2.1.2 Java Semantics of a Remotable Service

A remotable service is defined using the @Remotable annotation on the Java interface that defines the service. Remotable services are intended to be used for coarse grained services, and the parameters are passed by-value. Remotable Services MUST NOT make use of method overloading. \[JCA20001\]

The following snippet shows an example of a Java interface for a remote service:

```java
cpyackage services.hello;
@Remotable
public interface HelloService {
    String hello(String message);
}
```

2.1.3 Java Semantics of a Local Service

A local service can only be called by clients that are deployed within the same address space as the component implementing the local service.

A local interface is defined by a Java interface with no @Remotable annotation or it is defined by a Java class.

The following snippet shows an example of a Java interface for a local service:

```java
cpyackage services.hello;
public interface HelloService {
    String hello(String message);
}
```

The style of local interfaces is typically fine grained and is intended for tightly coupled interactions.

The data exchange semantic for calls to local services is by-reference. This means that implementation code which uses a local interface needs to be written with the knowledge that changes made to parameters (other than simple types) by either the client or the provider of the service are visible to the other.
2.1.4 @Reference

Accessing a service using reference injection is done by defining a field, a setter method, or a constructor parameter typed by the service interface and annotated with a @Reference annotation.

2.1.5 @Property

Implementations can be configured with data values through the use of properties, as defined in the SCA Assembly specification [ASSEMBLY]. The @Property annotation is used to define an SCA property.

2.2 Implementation Scopes: @Scope, @Init, @Destroy

Component implementations can either manage their own state or allow the SCA runtime to do so. In the latter case, SCA defines the concept of implementation scope, which specifies a visibility and lifecycle contract: an implementation has with the SCA runtime. Invocations on a service offered by a component will be dispatched by the SCA runtime to an implementation instance according to the semantics of its implementation scope.

Scopes are specified using the @Scope annotation on the implementation class.

This document defines two scopes:

- STATELESS
- COMPOSITE

Java-based implementation types can choose to support any of these scopes, and they can define new scopes specific to their type.

An implementation type can allow component implementations to declare lifecycle methods that are called when an implementation is instantiated or the scope is expired.

@Init denotes a method called upon first use of an instance during the lifetime of the scope (except for composite scoped implementation marked to eagerly initialize, see section Composite Scope). @Destroy specifies a method called when the scope ends.

Note that only no argument methods with a void return type can be annotated as lifecycle methods.

The following snippet is an example showing a fragment of a service implementation annotated with lifecycle methods:

```java
@Init
public void start() {
  ...
}

@Destroy
public void stop() {
  ...
}
```

The following sections specify the two standard scopes which a Java-based implementation type can support.

2.2.1 Stateless scope

For stateless scope components, there is no implied correlation between implementation instances used to dispatch service requests.
The concurrency model for the stateless scope is single threaded. This means that the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time. In addition, within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.

2.2.2 Composite scope

For a composite scope implementation instance, the SCA runtime MUST ensure that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite. The lifetime of the containing composite is defined as the time it becomes active in the runtime to the time it is deactivated, either normally or abnormally.

When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started. If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.

The concurrency model for the composite scope is multi-threaded. This means that the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.
3 Interface

This section describes the SCA Java interface element and the SCA metadata for Java interfaces.

3.1 Java interface element – <interface.java>

The Java interface element is used in SCDL files in places where an interface is declared in terms of a Java interface class. The Java interface element identifies the Java interface class and can also identify a callback interface, where the first Java interface represents the forward (service) call interface and the second interface represents the interface used to call back from the service to the client.

The *interface.java* element MUST conform to the schema defined in the `sca-interface-java.xsd` schema. [JCA30004]

The following is the pseudo-schema for the *interface.java* element

```
<interface.java interface="NCName" callbackInterface="NCName"/>
```

The *interface.java* element has the following attributes:

- **interface**: `NCName (1..1)` – the Java interface class to use for the service interface. The value of the `@interface` attribute MUST be the fully qualified name of the Java interface class. [JCA30001]
- **callbackInterface**: `NCName (0..1)` – the Java interface class to use for the callback interface. The value of the `@callbackInterface` attribute MUST be the fully qualified name of a Java interface used for callbacks. [JCA30002]

The following snippet shows an example of the Java interface element:

```
<interface.java interface="services.stockquote.StockQuoteService"
    callbackInterface="services.stockquote.StockQuoteServiceCallback"/>
```

Here, the Java interface is defined in the Java class file

`/services/stockquote/StockQuoteService.class`, where the root directory is defined by the contribution in which the interface exists. Similarly, the callback interface is defined in the Java class file

`/services/stockquote/StockQuoteServiceCallback.class`

Note that the Java interface class identified by the `@interface` attribute can contain a Java `@Callback` annotation which identifies a callback interface. If this is the case, then it is not necessary to provide the `@callbackInterface` attribute. However, if the Java interface class identified by the `@interface` attribute does contain a Java `@Callback` annotation, then the Java interface class identified by the `@callbackInterface` attribute MUST be the same interface class. [JCA30003]

For the Java interface type system, parameters and return types of the service methods are described using Java classes or simple Java types. It is recommended that the Java Classes used conform to the requirements of either JAXB [JAX-B] or of Service Data Objects [SDO] because of their integration with XML technologies.
3.2 @Remotable

The @Remotable annotation on a Java interface indicates that the interface is designed to be used for remote communication. Remotable interfaces are intended to be used for coarse-grained services. Operations' parameters and return values are passed by-value. Remotable Services are not allowed to make use of method overloading.

3.3 @Callback

A callback interface is declared by using a @Callback annotation on a Java service interface, with the Java Class object of the callback interface as a parameter. There is another form of the @Callback annotation, without any parameters, that specifies callback injection for a setter method or a field of an implementation.
4 Client API

This section describes how SCA services can be programmatically accessed from components and also from non-managed code, i.e. code not running as an SCA component.

4.1 Accessing Services from an SCA Component

An SCA component can obtain a service reference either through injection or programmatically through the ComponentContext API. Using reference injection is the recommended way to access a service, since it results in code with minimal use of middleware APIs. The ComponentContext API is provided for use in cases where reference injection is not possible.

4.1.1 Using the Component Context API

When a component implementation needs access to a service where the reference to the service is not known at compile time, the reference can be located using the component’s ComponentContext.

4.2 Accessing Services from non-SCA component implementations

This section describes how Java code not running as an SCA component that is part of an SCA composite accesses SCA services via references.

4.2.1 ComponentContext

Non-SCA client code can use the ComponentContext API to perform operations against a component in an SCA domain. How client code obtains a reference to a ComponentContext is runtime specific.

The following example demonstrates the use of the component Context API by non-SCA code:

```java
ComponentContext context = // obtained via host environment-specific means
HelloService helloService =
    context.getService(HelloService.class, "HelloService");
String result = helloService.hello("Hello World!");
```
5 Error Handling

Clients calling service methods can experience business exceptions and SCA runtime exceptions.

Business exceptions are thrown by the implementation of the called service method, and are defined as checked exceptions on the interface that types the service.

SCA runtime exceptions are raised by the SCA runtime and signal problems in management of component execution or problems interacting with remote services. The SCA runtime exceptions are defined in the Java API section.
Asynchronous Programming

Asynchronous programming of a service is where a client invokes a service and carries on executing without waiting for the service to execute. Typically, the invoked service executes at some later time. Output from the invoked service, if any, is fed back to the client through a separate mechanism, since no output is available at the point where the service is invoked. This is in contrast to the call-and-return style of synchronous programming, where the invoked service executes and returns any output to the client before the client continues. The SCA asynchronous programming model consists of:

- support for non-blocking method calls
- callbacks

Each of these topics is discussed in the following sections.

6.1 @OneWay

Nonblocking calls represent the simplest form of asynchronous programming, where the client of the service invokes the service and continues processing immediately, without waiting for the service to execute.

Any method with a void return type and which has no declared exceptions can be marked with a @OneWay annotation. This means that the method is non-blocking and communication with the service provider can use a binding that buffers the request and sends it at some later time.

For a Java client to make a non-blocking call to methods that either return values or which throw exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in section 9. It is considered to be a best practice that service designers define one-way methods as often as possible, in order to give the greatest degree of binding flexibility to deployers.

6.2 Callbacks

A callback service is a service that is used for asynchronous communication from a service provider back to its client, in contrast to the communication through return values from synchronous operations. Callbacks are used by bidirectional services, which are services that have two interfaces:

- an interface for the provided service
- a callback interface that is provided by the client

Callbacks can be used for both remotable and local services. Either both interfaces of a bidirectional service are remotable, or both are local. It is illegal to mix the two, as defined in the SCA Assembly specification [SCA Assembly].

A callback interface is declared by using a @Callback annotation on a service interface, with the Java Class object of the interface as a parameter. The annotation can also be applied to a method or to a field of an implementation, which is used in order to have a callback injected, as explained in the next section.

6.2.1 Using Callbacks

Bidirectional interfaces and callbacks are used when a simple request/response pattern isn’t sufficient to capture the business semantics of a service interaction. Callbacks are well suited for cases when a service request can result in multiple responses or new requests from the service back to the client, or where the service might respond to the client some time after the original request has completed.

The following example shows a scenario in which bidirectional interfaces and callbacks could be used. A client requests a quotation from a supplier. To process the enquiry and return the
quotation, some suppliers might need additional information from the client. The client does not
know which additional items of information will be needed by different suppliers. This interaction
can be modeled as a bidirectional interface with callback requests to obtain the additional
information.

```java
package somepackage;
import org.osoa.sca.annotation.Callback;
import org.osoa.sca.annotation.Remotable;
@Remotable
@Callback(QuotationCallback.class)
public interface Quotation {
    double requestQuotation(String productCode, int quantity);
}
```

```java
@Remotable
public interface QuotationCallback {
    String getState();
    String getZipCode();
    String getCreditRating();
}
```

In this example, the `requestQuotation` operation requests a quotation to supply a given quantity
of a specified product. The QuotationCallback interface provides a number of operations that the
supplier can use to obtain additional information about the client making the request. For
example, some suppliers might quote different prices based on the state or the zip code to which
the order will be shipped, and some suppliers might quote a lower price if the ordering company
has a good credit rating. Other suppliers might quote a standard price without requesting any
additional information from the client.

The following code snippet illustrates a possible implementation of the example service, using the
@Callback annotation to request that a callback proxy be injected.

```java
@Callback
protected QuotationCallback callback;

public double requestQuotation(String productCode, int quantity) {
    double price = getPrice(productQuote, quantity);
    double discount = 0;
    if (quantity > 1000 && callback.getState().equals("FL")) {
        discount = 0.05;
    }
    if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
        discount += 0.05;
    }
    return price * (1-discount);
}
```

The code snippet below is taken from the client of this example service. The client’s service
implementation class implements the methods of the QuotationCallback interface as well as those
of its own service interface ClientService.

```java
public class ClientImpl implements ClientService, QuotationCallback {
    private QuotationService myService;
    @Reference
    public void setMyService(QuotationService service) {
        myService = service;
    }
}
```
public void aClientMethod() {
    double quote = myService.requestQuotation("AB123", 2000);
    ...
}

public String getState() {
    return "TX";
}

public String getZipCode() {
    return "78746";
}

public String getCreditRating() {
    return "AA";
}

In this example the callback is **stateless**, i.e., the callback requests do not need any information relating to the original service request. For a callback that needs information relating to the original service request (a **stateful** callback), this information can be passed to the client by the service provider as parameters on the callback request.

### 6.2.2 Callback Instance Management

Instance management for callback requests received by the client of the bidirectional service is handled in the same way as instance management for regular service requests. If the client implementation has STATELESS scope, the callback is dispatched using a newly initialized instance. If the client implementation has COMPOSITE scope, the callback is dispatched using the same shared instance that is used to dispatch regular service requests.

As described in section 6.7.1, a stateful callback can obtain information relating to the original service request from parameters on the callback request. Alternatively, a composite-scoped client could store information relating to the original request as instance data and retrieve it when the callback request is received. These approaches could be combined by using a key passed on the callback request (e.g., an order ID) to retrieve information that was stored in a composite-scoped instance by the client code that made the original request.

### 6.2.3 Implementing Multiple Bidirectional Interfaces

Since it is possible for a single implementation class to implement multiple services, it is also possible for callbacks to be defined for each of the services that it implements. The service implementation can include an injected field for each of its callbacks. The runtime injects the callback onto the appropriate field based on the type of the callback. The following shows the declaration of two fields, each of which corresponds to a particular service offered by the implementation.

```java
@Callback
protected MyService1Callback callback1;

@Callback
protected MyService2Callback callback2;
```

If a single callback has a type that is compatible with multiple declared callback fields, then all of them will be set.
6.2.4 Accessing Callbacks

In addition to injecting a reference to a callback service, it is also possible to obtain a reference to a Callback instance by annotating a field or method of type `ServiceReference` with the `@Callback` annotation.

A reference implementing the callback service interface can be obtained using `ServiceReference.getService()`.

The following example fragments come from a service implementation that uses the callback API:

```java
@Callback
protected ServiceReference<MyCallback> callback;

public void someMethod() {
    MyCallback myCallback = callback.getCallback();    ...
    myCallback.receiveResult(theResult);
}
```

Because `ServiceReference` objects are serializable, they can be stored persistently and retrieved at a later time to make a callback invocation after the associated service request has completed. `ServiceReference` objects can also be passed as parameters on service invocations, enabling the responsibility for making the callback to be delegated to another service.

Alternatively, a callback can be retrieved programmatically using the `RequestContext` API. The snippet below shows how to retrieve a callback in a method programmatically:

```java
public void someMethod() {
    MyCallback myCallback =
        ComponentContext.getRequestContext().getCallback();
    ...
    myCallback.receiveResult(theResult);
}
```

On the client side, the service that implements the callback can access the callback ID that was returned with the callback operation by accessing the request context, as follows:

```java
@Context
protected RequestContext requestContext;

void receiveResult(Object theResult) {
    Object refParams =
        requestContext.getServiceReference().getCallbackID();
    ...
}
```

This is necessary if the service implementation has COMPOSITE scope, because callback injection is not performed for composite-scoped implementations.
7 Policy Annotations for Java

SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which influence how implementations, services, and references behave at runtime. The policy facilities are described in the SCA Policy Framework specification [POLICY]. In particular, the facilities include Intents and Policy Sets, where intents express abstract, high-level policy requirements and policy sets express low-level detailed concrete policies.

Policy metadata can be added to SCA assemblies through the means of declarative statements placed into Composite documents and into Component Type documents. These annotations are completely independent of implementation code, allowing policy to be applied during the assembly and deployment phases of application development.

However, it can be useful and more natural to attach policy metadata directly to the code of implementations. This is particularly important where the policies concerned are relied on by the code itself. An example of this from the Security domain is where the implementation code is expected to run under a specific security Role and where any service operations invoked on the implementation have to be authorized to ensure that the client has the correct rights to use the operations concerned. By annotating the code with appropriate policy metadata, the developer can rest assured that this metadata is not lost or forgotten during the assembly and deployment phases.

This specification has a series of annotations which provide the capability for the developer to attach policy information to Java implementation code. The annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to Java code. Secondly, there are further specific annotations that deal with particular policy intents for certain policy domains such as Security.

This specification supports using the Common Annotation for Java Platform specification (JSR-250) [JSR-250]. An implication of adopting the common annotation for Java platform specification is that the SCA Java specification supports consistent annotation and Java class inheritance relationships.

7.1 General Intent Annotations

SCA provides the annotation @Requires for the attachment of any intent to a Java class, to a Java interface or to elements within classes and interfaces such as methods and fields.

The @Requires annotation can attach one or multiple intents in a single statement.

Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI followed by the name of the Intent. The precise form used follows the string representation used by the javax.xml.namespace.QName class, which is as follows:

```
{" + Namespace URI + "} + intentname
```

Intents can be qualified, in which case the string consists of the base intent name, followed by a ".", followed by the name of the qualifier. There can also be multiple levels of qualification.

This representation is quite verbose, so we expect that reusable String constants will be defined for the namespace part of this string, as well as for each intent that is used by Java code. SCA defines constants for intents such as the following:

```
public static final String SCA_PREFIX = "http://docs.oasis-open.org/ns/opencsa/sca/200712";
public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
```

Notice that, by convention, qualified intents include the qualifier as part of the name of the constant, separated by an underscore. These intent constants are defined in the file that defines
an annotation for the intent (annotations for intents, and the formal definition of these constants, are covered in a following section).

Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

An example of the @Requires annotation with 2 qualified intents (from the Security domain) follows:

```java
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

This attaches the intents "confidentiality.message" and "integrity.message".

The following is an example of a reference requiring support for confidentiality:

```java
package com.foo;

import static org.oasisopen.sca.annotation.Confidentiality.*;
import static org.oasisopen.sca.annotation.Reference;
import static org.oasisopen.sca.annotation.Requires;

public class Foo {
    @Requires(CONFIDENTIALITY)
    @Reference
    public void setBar(Bar bar) {
        ...
    }
}
```

Users can also choose to only use constants for the namespace part of the QName, so that they can add new intents without having to define new constants. In that case, this definition would instead look like this:

```java
package com.foo;

import static org.oasisopen.sca.Constants.*;
import static org.oasisopen.sca.annotation.Reference;
import static org.oasisopen.sca.annotation.Requires;

public class Foo {
    @Requires(SCA_PREFIX+"confidentiality")
    @Reference
    public void setBar(Bar bar) {
        ...
    }
}
```

The formal syntax [EBNF-Syntax] for the @Requires annotation follows:

```
@Requires("QualifiedIntent ", "QualifiedIntent ")
```

where

```
QualifiedIntent ::= QName(" Qualified"
Qualifier ::= NCName
```

See section @Requires for the formal definition of the @Requires annotation.
7.2 Specific Intent Annotations

In addition to the general intent annotation supplied by the @Requires annotation described above, it is also possible to have Java annotations that correspond to specific policy intents. SCA provides a number of these specific intent annotations and it is also possible to create new specific intent annotations for any intent.

The general form of these specific intent annotations is an annotation with a name derived from the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an attribute to the annotation in the form of a string or an array of strings.

For example, the SCA confidentiality intent described in the section on General Intent Annotations using the @Requires(CONFIDENTIALITY) annotation can also be specified with the @Confidentiality specific intent annotation. The specific intent annotation for the "integrity" security intent is:

```java
@Integrity
```

An example of a qualified specific intent for the "authentication" intent is:

```java
@Authentication({ "message", "transport" })
```

This annotation attaches the pair of qualified intents: "authentication.message" and "authentication.transport" (the sca: namespace is assumed in both of these cases – "http://docs.oasis-open.org/ns/opencsa/sca/200712").

The general form of specific intent annotations is:

```
@Intent ("{ "qualifiers " })
```

where Intent is an NCName that denotes a particular type of intent.

7.2.1 How to Create Specific Intent Annotations

SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.

The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the String form of the QName of the intent. As part of the intent definition, it is good practice to also create String constants for the Namespace, for the Intent and for Qualified versions of the Intent (if defined). These String constants are then available for use with the @Requires annotation and it is also possible to use one or more of them as parameters to the specific intent annotation.

Alternatively, the QName of the intent can be specified using separate parameters for the targetNamespace and the localPart, for example:

```java
@Intent(targetNamespace=SCA_NS, localPart="confidentiality")
```

See section @Intent for the formal definition of the @Intent annotation.

When an intent can be qualified, it is good practice for the first attribute of the annotation to be a string (or an array of strings) which holds one or more qualifiers.

In this case, the attribute's definition needs to be marked with the @Qualifier annotation. The @Qualifier tells SCA that the value of the attribute is treated as a qualifier for the intent represented by the whole annotation. If more than one qualifier value is specified in an annotation, it means that multiple qualified forms exist. For example:

```java
@Confidentiality({"message", "transport"})
```

implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport" are set for the element to which the @Confidentiality annotation is attached.
See section `@Qualifier` for the formal definition of the `@Qualifier` annotation.

Examples of the use of the `@Intent` and the `@Qualifier` annotations in the definition of specific intent annotations are shown in the section dealing with Security Interaction Policy.

### 7.3 Application of Intent Annotations

The SCA Intent annotations can be applied to the following Java elements:

- Java class
- Java interface
- Method
- Field
- Constructor parameter

Where multiple intent annotations (general or specific) are applied to the same Java element, they are additive in effect. An example of multiple policy annotations being used together follows:

```
@Authentication
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

In this case, the effective intents are "authentication", "confidentiality.message" and "integrity.message".

If an annotation is specified at both the class/interface level and the method or field level, then the method or field level annotation completely overrides the class level annotation of the same base intent name.

The intent annotation can be applied either to classes or to class methods when adding annotated policy on SCA services. Applying an intent to the setter method in a reference injection approach allows intents to be defined at references.

### 7.3.1 Inheritance And Annotation

The inheritance rules for annotations are consistent with the common annotation specification, JSR 250 [JSR-250].

The following example shows the inheritance relations of intents on classes, operations, and super classes.

```java
package services.hello;

import org.oasisopen.sca.annotation.Remotable;
import org.oasisopen.sca.annotation.Integrity;
import org.oasisopen.sca.annotation.Authentication;

@Integrity("transport")
@Authentication
public class HelloService {
    @Integrity
    @Authentication("message")
    public String hello(String message) {...}

    @Integrity
    @Authentication("transport")
    public String helloThere() {...}
}
```

```java
package services.hello;

import org.oasisopen.sca.annotation.Remotable;
import org.oasisopen.sca.annotation.Confidentiality;
import org.oasisopen.sca.annotation.Authentication;
```

public class HelloChildService extends HelloService {

    @Confidentiality("transport")
    public String hello(String message) {...}

    @Authentication
    String helloWorld() {...}
}

Example 2a. Usage example of annotated policy and inheritance.

The effective intent annotation on the \texttt{helloWorld} method of the \texttt{HelloChildService} is \texttt{Integrity("transport")}, \texttt{@Authentication}, and \texttt{@Confidentiality("message")}.

The effective intent annotation on the \texttt{hello} method of the \texttt{HelloChildService} is \texttt{@Integrity("transport")}, \texttt{@Authentication}, and \texttt{@Confidentiality("transport")}, the same as in \texttt{HelloService} class.

The effective intent annotation on the \texttt{helloThere} method of the \texttt{HelloChildService} is \texttt{@Integrity} and \texttt{@Authentication("transport")}, the same as in \texttt{HelloService} class.

The effective intent annotation on the \texttt{hello} method of the \texttt{HelloService} is \texttt{@Integrity} and \texttt{@Authentication("message")}.

The listing below contains the equivalent declarative security interaction policy of the \texttt{HelloService} and \texttt{HelloChildService} implementation corresponding to the Java interfaces and classes shown in Example 2a.

```xml
<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    name="HelloServiceComposite">
    <service name="HelloService" requires="integrity/transport authentication">
        ...
    </service>
    <service name="HelloChildService" requires="integrity/transport authentication confidentiality/message">
        ...
    </service>
    ...

    <component name="HelloServiceComponent">
        <implementation java class="services.hello.HelloService"/>
        <operation name="hello" requires="integrity authentication/message"/>
        <operation name="helloThere" requires="integrity authentication/transport"/>
    </component>

    <component name="HelloChildServiceComponent">
        <implementation java class="services.hello.HelloChildService"/>
        <operation name="hello" requires="confidentiality/transport"/>
        <operation name="helloThere" requires="integrity/transport authentication"/>
        <operation name="helloWorld" requires="authentication"/>
    </component>
</composite>
```
Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

7.4 Relationship of Declarative And Annotated Intents

Annotated intents on a Java class cannot be overridden by declarative intents in a composite
document which uses the class as an implementation. This rule follows the general rule for intents
that they represent requirements of an implementation in the form of a restriction that cannot be
relaxed.

However, a restriction can be made more restrictive so that an unqualified version of an intent
expressed through an annotation in the Java class can be qualified by a declarative intent in a
using composite document.

7.5 Policy Set Annotations

The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies. For
example, a concrete policy is the specific encryption algorithm to use when encrypting messages
when using a specific communication protocol to link a reference to a service.

Policy Sets can be applied directly to Java implementations using the @PolicySets annotation.
The @PolicySets annotation either takes the QName of a single policy set as a string or the name
of two or more policy sets as an array of strings:

@PolicySets( "<policy set QName>" (, "<policy set QName>" )* )

As for intents, PolicySet names are QNames – in the form of "(Namespace-URI)localPart".

An example of the @PolicySets annotation:

    @Reference(name="helloService", required=true)
    @PolicySets( { MY_NS + "WS_Encryption_Policy",
                   MY_NS + "WS_Authentication_Policy" } )
    public setHelloService(HelloService service) { ... }

In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
using the namespace defined for the constant MY_NS.

PolicySets need to satisfy intents expressed for the implementation when both are present,
according to the rules defined in the Policy Framework specification [POLICY].

The SCA Policy Set annotation can be applied to the following Java elements:

- Java class
- Java interface
- Method
- Field
- Constructor parameter
7.6 Security Policy Annotations

This section introduces annotations for SCA's security intents, as defined in the SCA Policy Framework specification [POLICY].

7.6.1 Security Interaction Policy

The following interaction policy Intents and qualifiers are defined for Security Policy, which apply to the operation of services and references of an implementation:

- @Integrity
- @Confidentiality
- @Authentication

All three of these intents have the same pair of Qualifiers:

- message
- transport

The formal definitions of the @Authentication, @Confidentiality and @Integrity annotations are found in the sections @Authentication, @Confidentiality and @Integrity.

The following example shows an example of applying an intent to the setter method used to inject a reference. Accessing the hello operation of the referenced HelloService requires both "integrity.message" and "authentication.message" intents to be honored.

```java
package services.hello;
//Interface for HelloService
public interface HelloService {
    String hello(String helloMsg);
}

package services.client;
// Interface for ClientService
public interface ClientService {
    public void clientMethod();
}

// Implementation class for ClientService
package services.client;
public class ClientServiceImpl implements ClientService {
    private HelloService helloService;

    @Reference(name="helloService", required=true)
    @Integrity("message")
    @Authentication("message")
    public void setHelloService(HelloService service) {
        helloService = service;
    }

    public void clientMethod() {
        String result = helloService.hello("Hello World!");
    }
```
Example 1. Usage of annotated intents on a reference.

7.6.2 Security Implementation Policy

SCA defines a number of security policy annotations that apply as policies to implementations themselves. These annotations mostly have to do with authorization and security identity. The following authorization and security identity annotations (as defined in JSR 250) are supported:

- **RunAs**
  - Takes as a parameter a string which is the name of a Security role.
  - eg. @RunAs("Manager")

- **RolesAllowed**
  - Takes as a parameter a single string or an array of strings which represent one or more role names. When present, the implementation can only be accessed by principals whose role corresponds to one of the role names listed in the @roles attribute. How role names are mapped to security principals is implementation dependent (SCA does not define this).
  - eg. @RolesAllowed( {"Manager", "Employee"} )

- **PermitAll**
  - No parameters. When present, grants access to all roles.

- **DenyAll**
  - No parameters. When present, denies access to all roles.

- **DeclareRoles**
  - Takes as a parameter a string or an array of strings which identify one or more role names that form the set of roles used by the implementation.
  - eg. @DeclareRoles("Manager", "Employee", "Customer")

(all these are declared in the Java package javax.annotation.security)

For a full explanation of these intents, see the Policy Framework specification [POLICY].

7.6.2.1 Annotated Implementation Policy Example

The following is an example showing annotated security implementation policy:

```
package services.account;
@Remotable
public interface AccountService {
    AccountReport getAccountReport(String customerID);
    float fromUSDollarToCurrency(float value);
}
```

The following is a full listing of the AccountServiceImpl class, showing the Service it implements, plus the service references it makes and the settable properties that it has, along with a set of implementation policy annotations:

```
package services.account;
```
import java.util.List;
import commonj.sdo.DataFactory;
import org.oasisopen.sca.annotation.Property;
import org.oasisopen.sca.annotation.Reference;
import org.oasisopen.sca.annotation.RolesAllowed;
import org.oasisopen.sca.annotation.RunAs;
import org.oasisopen.sca.annotation.PermitAll;
import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;

@RolesAllowed("customers")
@RunAs("accountants")
public class AccountServiceImpl implements AccountService {

    @Property
    protected String currency = "USD";

    @Reference
    protected AccountDataService accountDataService;
    @Reference
    protected StockQuoteService stockQuoteService;

    @RolesAllowed({"customers", "accountants"})

    public AccountReport getAccountReport(String customerId) {
        DataFactory dataFactory = DataFactory.INSTANCE;
        AccountReport accountReport =
            (AccountReport)dataFactory.create(AccountReport.class);
        List accountSummaries = accountReport.getAccountSummaries();

        CheckingAccount checkingAccount =
            accountDataService.getCheckingAccount(customerId);
        AccountSummary checkingAccountSummary =
            (AccountSummary)dataFactory.create(AccountSummary.class);
        checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
        checkingAccountSummary.setAccountType("checking");
        checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
        accountSummaries.add(checkingAccountSummary);

        SavingsAccount savingsAccount =
            accountDataService.getSavingsAccount(customerId);
        AccountSummary savingsAccountSummary =
            (AccountSummary)dataFactory.create(AccountSummary.class);
        savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
        savingsAccountSummary.setAccountType("savings");
        savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
        accountSummaries.add(savingsAccountSummary);

        StockAccount stockAccount =
            accountDataService.getStockAccount(customerId);
        AccountSummary stockAccountSummary =
            (AccountSummary)dataFactory.create(AccountSummary.class);
    }
}
(AccountSummary)dataFactory.create(AccountSummary.class);
stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
stockAccountSummary.setAccountType("stock");
float balance = (stockQuoteService.getQuote(stockAccount.getSymbol())) *
stockAccount.getQuantity();
stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
accountSummaries.add(stockAccountSummary);

return accountReport;
}

@PermitAll
public float fromUSDollarToCurrency(float value) {
    if (currency.equals("USD")) return value;
    if (currency.equals("EURO")) return value * 0.8f;
    return 0.0f;
}

Example 3. Usage of annotated security implementation policy for the java language.
In this example, the implementation class as a whole is marked:
• @RolesAllowed("customers") - indicating that customers have access to the
  implementation as a whole
• @RunAs("accountants") - indicating that the code in the implementation runs with the
  permissions of accountants
The getAccountReport(..) method is marked with @RolesAllowed(\{"customers", "accountants"\}),
which indicates that this method can be called by both customers and accountants.
The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
can be called by any role.
8 Java API

This section provides a reference for the Java API offered by SCA.

8.1 Component Context

The following Java code defines the ComponentContext interface:

```java
package org.oasisopen.sca;

public interface ComponentContext {
    String getURI();

    <B> B getService(Class<B> businessInterface, String referenceName);

    <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
                                                    String referenceName);

    <B> Collection<B> getServices(Class<B> businessInterface,
                                   String referenceName);

    // Collection<ServiceReference<B>> getServiceReferences(Class<B> businessInterface,
                                                             String referenceName);

    <B> ServiceReference<B> createSelfReference(Class<B> businessInterface);

    <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
                                               String serviceName);

    <B> B getProperty(Class<B> type, String propertyName);

    <B, R extends ServiceReference<B>> R cast(B target)
      throws IllegalArgumentException;

    RequestContext getRequestContext();
}
```

- **getURI** - returns the absolute URI of the component within the SCA domain
- **getService(Class<B> businessInterface, String referenceName)** - Returns a proxy for the reference defined by the current component. The getService() method takes as its input arguments the Java type used to represent the target service on the client and the name of the service reference. It returns an object providing access to the service. The returned object implements the Java interface the service is typed with.
- **getServiceImpl(Class<B> businessInterface, String referenceName)** – Returns a ServiceReference defined by the current component. This method MUST throw an IllegalArgumentException if the reference has multiplicity greater than one.
• `getServices(Class<B> businessInterface, String referenceName)` – Returns a list of typed service proxies for a business interface type and a reference name.

• `getServiceReferences(Class<B> businessInterface, String referenceName)` – Returns a list typed service references for a business interface type and a reference name.

• `createSelfReference(Class<B> businessInterface)` – Returns a `ServiceReference` that can be used to invoke this component over the designated service.

• `createSelfReference(Class<B> businessInterface, String serviceName)` – Returns a `ServiceReference` that can be used to invoke this component over the designated service.
  Service name explicitly declares the service name to invoke

• `getProperty(Class<B> type, String propertyName)` – Returns the value of an SCA property defined by this component.

• `getRequestContext()` – Returns the context for the current SCA service request, or null if there is no current request or if the context is unavailable. The `ComponentContext.getRequestContext` method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.

• `cast(B target)` – Casts a type-safe reference to a `ServiceReference`.

A component can access its component context by defining a field or setter method typed by `org.oasisopen.sca.ComponentContext` and annotated with `@Context`. To access the target service, the component uses `ComponentContext.getService(..)`.

The following shows an example of component context usage in a Java class using the `@Context` annotation.

```java
private ComponentContext componentContext;

@Context
public void setContext(ComponentContext context) {
    componentContext = context;
}

public void doSomething() {
    HelloWorld service =
        componentContext.getService(HelloWorld.class, "HelloWorldComponent");
    service.hello("hello");
}
```

Similarly, non-SCA client code can use the `ComponentContext` API to perform operations against a component in an SCA domain. How the non-SCA client code obtains a reference to a `ComponentContext` is runtime specific.

### 8.2 Request Context

The following shows the `RequestContext` interface:

```java
package org.oasisopen.sca;

import javax.security.auth.Subject;

public interface RequestContext {
    Subject getSecuritySubject();
}
```
The RequestContext interface has the following methods:

- **getSecuritySubject()** – Returns the JAAS Subject of the current request.
- **getServiceName()** – Returns the name of the service on the Java implementation the request came in on.
- **getCallbackReference()** – Returns a service reference to the callback as specified by the caller. This method returns null when called for a service request whose interface is not bidirectional or when called for a callback request.
- **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the getCallbackReference() method, this method returns null when called for a service request whose interface is not bidirectional or when called for a callback request.
- **getCallbackReference()** – When invoked during the execution of a service operation, the getCallbackReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getCallbackReference method MUST return a ServiceReference that represents the callback that was invoked. [JCA80003]

### 8.3 ServiceReference

ServiceReferences can be injected using the @Reference annotation on a field, a setter method, or constructor parameter taking the type ServiceReference. The detailed description of the usage of these methods is described in the section on Asynchronous Programming in this document.

The following Java code defines the ServiceReference interface:

```java
package org.oasisopen.sca;

public interface ServiceReference<B> extends java.io.Serializable {
    B getService();
    Class<B> getBusinessInterface();
}
```

The ServiceReference interface has the following methods:

- **getService()** - Returns a type-safe reference to the target of this reference. The instance returned is guaranteed to implement the business interface for this reference. The value returned is a proxy to the target that implements the business interface associated with this reference.
- **getBusinessInterface()** – Returns the Java class for the business interface associated with this reference.

### 8.4 ServiceRuntimeException

The following snippet shows the ServiceRuntimeException.

```java
package org.oasisopen.sca;
```
public class ServiceRuntimeException extends RuntimeException {
    --
}

This exception signals problems in the management of SCA component execution.

8.5 ServiceUnavailableException

The following snippet shows the ServiceUnavailableException.

package org.oasisopen.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    --
}

This exception signals problems in the interaction with remote services. These are exceptions that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException that is not a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since it most likely requires human intervention.

8.6 InvalidServiceException

The following snippet shows the InvalidServiceException.

package org.oasisopen.sca;

public class InvalidServiceException extends ServiceRuntimeException {
    --
}

This exception signals that the ServiceReference is no longer valid. This can happen when the target of the reference is undeployed. This exception is not transient and therefore is unlikely to be resolved by retrying the operation and will most likely require human intervention.

8.7 Constants

The SCA Constants interface defines a number of constant values that are used in the SCA Java APIs and Annotations. The following snippet shows the Constants interface:

package org.oasisopen.sca;

public interface Constants {
    String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
    String SCA_PREFIX = "{+SCA_NS+}";
}
9 Java Annotations

This section provides definitions of all the Java annotations which apply to SCA.

This specification places constraints on some annotations that are not detectable by a Java compiler. For example, the definition of the @Property and @Reference annotations indicate that they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to constructor parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code. [JCA90001]

SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class. [JCA90002]

9.1 @AllowsPassByReference

The following Java code defines the @AllowsPassByReference annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface AllowsPassByReference {

// The @AllowsPassByReference annotation is used on implementations of remotable interfaces to indicate that interactions with the service from a client within the same address space are allowed to use pass by reference data exchange semantics. The implementation promises that its by-value semantics will be maintained even if the parameters and return values are actually passed by reference. This means that the service will not modify any operation input parameter or return value, even after returning from the operation. Either a whole class implementing a remotable service or an individual remotable service method implementation can be annotated using the @AllowsPassByReference annotation.

@AllowsPassByReference has no attributes

The following snippet shows a sample where @AllowsPassByReference is defined for the implementation of a service method on the Java component implementation class.

@AllowsPassByReference
public String hello(String message) {
    ...
}
```

9.2 @Authentication

The following Java code defines the @Authentication annotation:
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import static org.oasisopen.sca.Constants.SCA_PREFIX;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Inherited
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
@Intent(Authentication.AUTHENTICATION)
public @interface Authentication {
    String AUTHENTICATION = SCA_PREFIX + "authentication";
    String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
    String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";

    /**
     * List of authentication qualifiers (such as "message"
     * or "transport")
     * @return authentication qualifiers
     */
    @Qualifier
    String[] value() default "";
}

The @Authentication annotation is used to indicate that the invocation requires authentication. See the section on Application of Intent Annotations for samples and details.

9.3 @Callback

The following Java code defines the @Callback annotation:

The @Callback annotation is used to annotate a service interface with a callback interface by specifying the Java class object of the callback interface as an attribute.

The @Callback annotation has the following attribute:

- **value** - the name of a Java class file containing the callback interface

The @Callback annotation can also be used to annotate a method or a field of an SCA implementation class, in order to have a callback object injected. When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes. [JCA90046]

An example use of the @Callback annotation to declare a callback interface follows:

```java
tag:1201
package somepackage;
tag:1202
import org.oasisopen.sca.annotation.Callback;
tag:1203
import org.oasisopen.sca.annotation.Remotable;
tag:1204
@Remotable
@Callback(MyServiceCallback.class) tag:1205
public interface MyService {
tag:1206
    void someMethod(String arg);
tag:1207
}
tag:1208
@Remotable
public interface MyServiceCallback {
tag:1209
    void receiveResult(String result);
tag:1210
}
tag:1211
```

In this example, the implied component type is:

```xml
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
tag:1212
    <service name="MyService">
tag:1213
        <interface.java interface="somepackage.MyService" tag:1214
            callbackInterface="somepackage.MyServiceCallback"/>
tag:1215
    </service>
tag:1216
</componentType>
tag:1217
```

9.4 @ComponentName

The following Java code defines the @ComponentName annotation:

```java
package org.oasisopen.sca.annotation;
tag:1236
import static java.lang.annotation.ElementType.METHOD;
tag:1237
import static java.lang.annotation.ElementType.FIELD;
tag:1238
import static java.lang.annotation.RetentionPolicy.RUNTIME;
tag:1239
import java.lang.annotation.Retention;
tag:1240
import java.lang.annotation.Target;
tag:1241

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ComponentName {
tag:1248
}
tag:1251
```

```
tag:1202 print:Deleted: a callback interface, which takes the Java Class object of the callback interface as a parameter.
tag:1204
Formatted: Font color: Red
Deleted: ¶
tag:1208
Deleted: ¶
tag:1209
Deleted: ¶
tag:1211
Deleted: ¶
tag:1217
Deleted: ¶
tag:1222
Deleted: ¶
tag:1224
Deleted: ¶
tag:1229
Deleted: ¶
tag:1231
Deleted: ¶
tag:1237
Deleted: ¶
tag:1241
Deleted: ¶
tag:1244
Deleted: ¶
tag:1248
Deleted: ¶
tag:1251
```
The @ComponentName annotation is used to denote a Java class field or setter method that is used to inject the component name.

The following snippet shows a component name field definition sample.

```java
@ComponentName
private String componentName;
```

The following snippet shows a component name setter method sample.

```java
@ComponentName
public void setComponentName(String name) {
    /*...
```

### 9.5 @Confidentiality

The following Java code defines the `@Confidentiality` annotation:

```java
package org.oasisopen.sca.annotations;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import static org.oasisopen.sca.Constants.SCA_PREFIX;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Inherited
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
@Intent(Confidentiality.CONFIDENTIALITY)
public @interface Confidentiality {
    String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
    String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
    String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";

    /**
     * List of confidentiality qualifiers such as "message" or "transport".
     * 
     * @return confidentiality qualifiers
     */
    @Qualifier
    String[] value() default "";
}
```

The `@Confidentiality` annotation is used to indicate that the invocation requires confidentiality.

See the section on Application of Intent Annotations for samples and details.
9.6 @Constructor

The following Java code defines the @Constructor annotation:

```java
package org.oasisopen.sca.annotation;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(CONSTRUCTOR)
@Retention(RUNTIME)
public @interface Constructor {}
```

The @Constructor annotation is used to mark a particular constructor to use when instantiating a Java component implementation. If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation. [JCA90003]

The following snippet shows a sample for the @Constructor annotation.

```java
public class HelloServiceImpl implements HelloService {
    public HelloServiceImpl(){
        ...
    }
    @Constructor
    public HelloServiceImpl(@Property(name="someProperty") String someProperty ){
        ...
    }
    public String hello(String message) {
        ...
    }
}
```

9.7 @Context

The following Java code defines the @Context annotation:

```java
package org.oasisopen.sca.annotation;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Context {
```

Comment [MES]: There also needs to be a normative statement that at most 1 constructor can be annotated with @Constructor.

Deleted: If this constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation.
The @Context annotation is used to denote a Java class field or a setter method that is used to inject a composite context for the component. The type of context to be injected is defined by the type of the Java class field or type of the setter method input argument; the type is either ComponentContext or RequestContext.

The @Context annotation has no attributes.

The following snippet shows a ComponentContext field definition sample.

```java
@Context
protected ComponentContext context;
```

The following snippet shows a RequestContext field definition sample.

```java
@Context
protected RequestContext context;
```

### 9.8 @Destroy

The following Java code defines the @Destroy annotation:

```java
class org.oasisopen.sca.annotation;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface Destroy {
    ...
}
```

The @Destroy annotation is used to denote a single Java class method that will be called when the scope defined for the implementation class ends. A method annotated with @Destroy MAY have any access modifier and MUST have a void return type and no arguments. If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends. If the implementation class has a method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST NOT instantiate the implementation class.

```java
@Destroy
public void myDestroyMethod() {
    ...
}
```

### 9.9 @EagerInit

The following Java code defines the @EagerInit annotation:
The @EagerInit annotation is used to mark the Java class of a COMPOSITE scoped implementation for eager initialization. When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started. [JCA90007]

9.10 @Init

The following Java code defines the @Init annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface Init {
  }
```

The @Init annotation is used to denote a single Java class method that is called when the scope defined for the implementation class starts. A method marked with the @Init annotation MAY have any access modifier and MUST have a void return type and no arguments. If there is a method that matches the criteria for the @Init annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete. [JCA90009]

The following snippet shows an example of an init method definition.

```java
public void myInitMethod() {
  ...
}
```

9.11 @Integrity

The following Java code defines the @Integrity annotation:

```java
package org.oasisopen.sca.annotation;
```

Deleted: annotate
Formatted: Pattern: Clear (Yellow)
Deleted: When marked for eager initialization, the composite scoped instance is created when its containing component is started.
Formatted: Font color: Red
Deleted: The
Formatted: Pattern: Clear (Yellow)
Deleted: method MAY have any access modifier and MUST have a void return type and no arguments.
Formatted: Font color: Red
Deleted: If there is a method that matches these criteria, the SCA runtime MUST call the annotated method after all property and reference injection is complete.
Formatted: Font color: Red
Deleted: If the implementation class has a method with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT instantiate the implementation class.
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import static org.oasisopen.Constants.SCA_PREFIX;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Inherited
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
@Intent(Integrity.INTEGRITY)
public @interface Integrity {
    String INTEGRITY = SCA_PREFIX + "integrity";
    String INTEGRITY_MESSAGE = INTEGRITY + ".message";
    String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";

    /**
     * List of integrity qualifiers (such as "message" or "transport").
     * @return integrity qualifiers
     */
    @Qualifier
    String[] value() default "";
}

The @Integrity annotation is used to indicate that the invocation requires integrity (ie no tampering of the messages between client and service).

See the section on Application of Intent Annotations for samples and details.

9.12 @Intent

The following Java code defines the @Intent annotation:

package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface Intent {
    /**
     * The qualified name of the intent, in the form defined by
     * {@link javax.xml.namespace.QName#toString}.
     * @return the qualified name of the intent
     */
    String value() default "";

    /**
     * The XML namespace for the intent.
     */
}
The @Intent annotation is used for the creation of new annotations for specific intents. It is not expected that the @Intent annotation will be used in application code. See the section "How to Create Specific Intent Annotations" for details and samples of how to define new intent annotations.

9.13 @OneWay

The following Java code defines the @OneWay annotation:

```java
package org.oasisopen.sca.annotation;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface OneWay {
    String targetNamespace() default "";
    String localPart() default "";
}

The @OneWay annotation is used on a Java interface or class method to indicate that invocations will be dispatched in a non-blocking fashion as described in the section on Asynchronous Programming.

The @OneWay annotation has no attributes.

The following snippet shows the use of the @OneWay annotation on an interface.

```java
package services.hello;

import org.oasisopen.sca.annotation.OneWay;

public interface HelloService {
    @OneWay
    void hello(String name);
}
```

Comment [ME6]: Needs recasting in a normative form of statement

9.14 @PolicySets

The following Java code defines the @PolicySets annotation:

```java
package org.oasisopen.sca.annotation;
```
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
public @interface PolicySets {

/**
 * Returns the policy sets to be applied.
 */

   * @return the policy sets to be applied
   */
    String[] value() default "";
    }

The @PolicySets annotation is used to attach one or more SCA Policy Sets to a Java implementation class or to one of its subelements.

See the section "Policy Set Annotations" for details and samples.

9.15 @Property

The following Java code defines the @Property annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Property {

    String name() default "";
    boolean required() default true;
}
```

The @Property annotation is used to denote a Java class field, a setter method, or a constructor parameter that is used to inject an SCA property value. The type of the property injected, which can be a simple Java type or a complex Java type, is defined by the type of the Java class field or the type of the input parameter of the setter method or constructor.

The @Property annotation can be used on fields, on setter methods or on a constructor method parameter. However, the @Property annotation MUST NOT be used on a class field that is declared as final. [JCA90011]

Properties can also be injected via setter methods even when the @Property annotation is not present. However, the @Property annotation MUST be used in order to inject a property onto a non-public field. [JCA90012] In the case where there is no @Property annotation, the name of the property is the same as the name of the field or setter.
Where there is both a setter method and a field for a property, the setter method is used.

The @Property annotation has the following attributes:

- **name (optional)** – the name of the property. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present. [JCA90013]

- **required (optional)** – a boolean value which specifies whether injection of the property value is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Property annotation applied to a constructor parameter, the required attribute MUST have the value true. [JCA90014]

The following snippet shows a property field definition sample.

```java
@Property(name="currency", required=true)
protected String currency;
```

The following snippet shows a property setter sample

```java
@Property(name="currency", required=true)
public void setCurrency(String theCurrency) {
    ....
}
```

For a @Property annotation, if the property is defined as an array or as any type that extends or implements java.util.Collection, then the implied component type of the implementation MUST have a property with a many attribute set to true, otherwise @many is set to false. [JCA90047]

The following snippet shows the definition of a configuration property using the @Property annotation for a collection.

```java
private List<String> helloConfigurationProperty;
```

```java
@Property(required=true)
public void setHelloConfigurationProperty(List<String> property) {
    helloConfigurationProperty = property;
}
```

### 9.16 @Qualifier

The following Java code defines the @Qualifier annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
```
The @Qualifier annotation is applied to an attribute of a specific intent annotation definition, defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers. [JCA90015]

See the section "How to Create Specific Intent Annotations" for details and samples of how to define new intent annotations.

9.17 @Reference

The following Java code defines the @Reference annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Reference {
    String name() default "";
    boolean required() default true;
}
```

The @Reference annotation type is used to annotate a Java class field, a setter method, or a constructor parameter that is used to inject a service that resolves the reference. The interface of the service injected is defined by the type of the Java class field or the type of the input parameter of the setter method or constructor.

The @Reference annotation MUST NOT be used on a class field that is declared as final. [JCA90016]

References can also be injected via setter methods even when the @Reference annotation is not present. However, the @Reference annotation MUST be used in order to inject a reference onto a non-public field. [JCA90017] In the case where there is no @Reference annotation, the name of the reference is the same as the name of the field or setter.

Where there is both a setter method and a field for a reference, the setter method is used.

The @Reference annotation has the following attributes:

- **name** : String (optional) – the name of the reference. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name corresponding to the setter method name. For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present. [JCA90018]
• **required (optional)** – a boolean value which specifies whether injection of the service reference is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true. [JCA90019]

The following snippet shows a reference field definition sample.

```java
@Reference(name="stockQuote", required=true)
protected StockQuoteService stockQuote;
```

The following snippet shows a reference setter sample

```java
@Reference(name="stockQuote", required=true)
public void setStockQuote( StockQuoteService theSQService ) {
    ...
}
```

The following fragment from a component implementation shows a sample of a service reference using the @Reference annotation. The name of the reference is “helloService” and its type is HelloService. The clientMethod() calls the “hello” operation of the service referenced by the helloService reference.

```java
package services.hello;

private HelloService helloService;

@Reference(name="helloService", required=true)
public setHelloService(HelloService service) {
    helloService = service;
}

public void clientMethod() {
    String result = helloService.hello("Hello World!");
    ...
}
```

The presence of a @Reference annotation is reflected in the componentType information that the runtime generates through reflection on the implementation class. The following snippet shows the component type for the above component implementation fragment.

```xml
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
    <!-- Any services offered by the component would be listed here -->
    <interface.java interface="services.hello.HelloService"/>
</componentType>
```
If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..1 if the @Reference annotation required attribute is true. [JCA90020]

If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true. [JCA90021]

The following fragment from a component implementation shows a sample of a service reference definition using the @Reference annotation on a java.util.List. The name of the reference is "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the services referenced by the helloServices reference. In this case, at least one HelloService needs to be present, so **required** is true.

```java
@Reference(name="helloServices", required=true)
protected List<HelloService> helloServices;

public void clientMethod() {
    ... for (int index = 0; index < helloServices.size(); index++) {
        HelloService helloService = (HelloService)helloServices.get(index);
        String result = helloService.hello("Hello World!");
    }
    ...
}
```

The following snippet shows the XML representation of the component type reflected from for the former component implementation fragment. There is no need to author this component type in this case since it can be reflected from the Java class.

```xml
<componentType>
    <reference name="helloServices" multiplicity="1..n"/>
    <interface java interface="services.hello.HelloService"/>
</componentType>
```

An unwired reference with a multiplicity of 0..1 **MUST** be presented to the implementation code by the SCA runtime as null. [JCA90022] An unwired reference with a multiplicity of 0..n **MUST** be presented to the implementation code by the SCA runtime as an empty array or empty collection. [JCA90023]

### 9.17.1 Reinjection

References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized. [JCA90024]

In order for reinjection to occur, the following MUST be true:

- References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized. [JCA90024]
- An unwired reference with a multiplicity of 0..1 **MUST** be presented to the implementation code by the SCA runtime as null. [JCA90022] An unwired reference with a multiplicity of 0..n **MUST** be presented to the implementation code by the SCA runtime as an empty array or empty collection. [JCA90023]

### Deleted

- If the reference is not an array or collection, then the implied component type has a reference with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference required attribute – 1..1 applies if required=true.
- If the reference is defined as an array or as any type that extends or implements java.util.Collection, then the implied component type has a reference with a multiplicity of either 1..n or 0..n.

### Formatted

- Font color: Red
- Font: Not Bold, Not Italic, Complex Script

### Deleted

- Depending on whether the required attribute of the @Reference annotation is set to true or false – 1..n applies if required=true.
- At runtime, the representation of an unwired reference depends on the reference's multiplicity. An unwired reference with a multiplicity of 0..1 must be null.

### Formatted

- Font color: Red
- Font: Not Bold, Not Italic

### Deleted

- An unwired reference with a multiplicity of 0..1 **MUST** be presented to the implementation code by the SCA runtime as null. [JCA90022] An unwired reference with a multiplicity of 0..n **MUST** be presented to the implementation code by the SCA runtime as an empty array or empty collection. [JCA90023]

### Formatted

- Font color: Red
- Font: Not Bold, Not Italic

### Deleted

- References MAY be reinjected after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.

### Formatted

- Font color: Red
- Font: Not Bold, Not Italic

### Deleted

- References MAY be reinjected after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.
1. The component MUST NOT be STATELESS scoped.

2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed.

   [JCA90025]

    Setter injection allows for code in the setter method to perform processing in reaction to a change.

    If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed. [JCA90026]

    If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made. [JCA90029] If it doesn’t work, the exception thrown will depend on the runtime and the cause of the failure.

A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed. [JCA90030] If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException when an operation is invoked on the ServiceReference. [JCA90031] If the target service of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference. [JCA90032] If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made. [JCA90033] If it doesn’t work, the exception thrown will depend on the runtime and the cause of the failure.

A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place. If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the deployed or unavailable service, and attempts to call business methods should throw an InvalidServiceException or a ServiceUnavailableException. [JCA90034] If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service. [JCA90035]

The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This means that in the cases where reference reinjection is not allowed, the array or Collection for a reference with multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring. [JCA90036] In cases where the contents of a reference array or collection MAY change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called for any change to the contents. [JCA90038] A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component. [JCA90039]

**Effect on**

<table>
<thead>
<tr>
<th>Change event</th>
<th>Injected Reference or ServiceReference</th>
<th>Existing ServiceReference Object**</th>
<th>Subsequent invocations of ComponentContext.getServiceReference() or getService()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change to the target of the reference <strong>can be reinjected</strong> (if other conditions* apply). If not reinjected, then it continues to work as if the reference target was not changed.</td>
<td>continue to work as if the reference target was not changed.</td>
<td>Result corresponds to the current configuration of the domain.</td>
<td></td>
</tr>
</tbody>
</table>
the reference target was not changed.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>undeployed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target service</th>
<th>Business methods throw ServiceUnavailableException</th>
<th>Business methods throw ServiceUnavailableException</th>
<th>Result is be a reference to the unavailable service. Business methods throw ServiceUnavailableException.</th>
</tr>
</thead>
<tbody>
<tr>
<td>becomes unavailable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target service</th>
<th>might continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.</th>
<th>might continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.</th>
<th>Result is a reference to the changed service.</th>
</tr>
</thead>
<tbody>
<tr>
<td>changed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Other conditions:

1. The component MUST NOT be STATELESS scoped.
2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed.

** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().

### 9.18 @Remotable

The following Java code defines the @Remotable annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Remotable {
}
```

The @Remotable annotation is used to specify a Java service interface as remotable. A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.

The @Remotable annotation has no attributes.

The following snippet shows the Java interface for a remotable service with its @Remotable annotation.
The style of remotable interfaces is typically coarse grained and intended for loosely coupled interactions. Remotable service interfaces are not allowed to make use of method overloading. Complex data types exchanged via remotable service interfaces need to be compatible with the marshalling technology used by the service binding. For example, if the service is going to be exposed using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types or they can be Service Data Objects (SDOs) [SDO]. Independent of whether the remotable service is called from outside of the composite that contains it or from another component in the same composite, the data exchange semantics are by-value. Implementations of remotable services can modify input data during or after an invocation and can modify return data after the invocation. If a remotable service is called locally or remotely, the SCA container is responsible for making sure that no modification of input data or post-invocation modifications to return data are seen by the caller.

The following snippet shows a remotable Java service interface.

```java
package services.hello;

import org.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {
    String hello(String message);
}
```

9.19 @Requires

The following Java code defines the @Requires annotation:

```java
package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.FIELD;
```
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Inherited
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Requires {

    /**
     * Returns the attached intents.
     * *
     * @return the attached intents
     */
    String[] value() default "";

}

The @Requires annotation supports general purpose intents specified as strings. Users can also define specific intent annotations using the @Intent annotation. See the section "General Intent Annotations" for details and samples.

9.20 @Scope

The following Java code defines the @Scope annotation:

package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Scope {

    String value() default "STATELESS";

}

The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface. [JCA90041]

The @Scope annotation has the following attribute:

- **value** - the name of the scope.

SCA defines the following scope names, but others can be defined by particular Java-based implementation types:

STATELESS

COMPOSITE

For 'STATELESS' implementations, a different implementation instance can be used to service each request. Implementation instances can be newly created or be drawn from a pool of instances.

The default value is STATELESS.

The following snippet shows a sample for a COMPOSITE scoped service implementation:

package services.hello;
import org.oasisopen.sca.annotation.*;

@Service(HelloService.class)
@Scope("COMPOSITE")
public class HelloServiceImpl implements HelloService {
  public String hello(String message) {
    ...
  }
}

9.21 @Service

The following Java code defines the @Service annotation:

package org.oasisopen.sca.annotation;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Service {
  Class<?>[] interfaces() default {};
  Class<?> value() default Void.class;
}

The @Service annotation is used on a component implementation class to specify the SCA services offered by the implementation. An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present. A class used as the implementation of a service is not required to have a @Service annotation. If a class has no @Service annotation, then the rules determining which services are offered and what interfaces those services have are determined by the specific implementation type.

The @Service annotation has the following attributes:

- interfaces (1..1) – The value is an array of interface or class objects that are exposed as services by this component implementation.
- value – A shortcut for the case when the class provides only a single service interface - contains a single interface or class object that is exposed as a service by this component implementation.

A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.

A @Service annotation with no attributes MUST be ignored, it is the same as not having the annotation there at all.

The service names of the defined services default to the names of the interfaces or class, without the package name.
A component implementation MUST NOT have two services with the same Java simple name.

If a Java implementation needs to realize two services with the same Java simple name then this can be achieved through subclassing of the interface.

The following snippet shows an implementation of the HelloService marked with the @Service annotation.

```java
package services.hello;

import org.oasisopen.sca.annotation.Service;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {
    public void hello(String name) {
        System.out.println("Hello "+ name);
    }
}
```

Deleted: A component MUST NOT have two services with the same Java simple name.
10 WSDL to Java and Java to WSDL

The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotaible Java interfaces from WSDL portTypes and vice versa.

For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn’t. [JCA100001] The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation. [JCA100002] For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable. [JCA100003]

For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B] mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MAY support the SDO 2.1 mapping from Java types to XML schema types. [JCA100004] SCA runtimes MUST support the SDO 2.1 mapping from Java types to XML schema types. [JCA100005] Having a choice of binding technologies is allowed, as noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is referenced by the JAX-WS specification.

The JAX-WS mappings are applied with the following restrictions:

- No support for holders

Note: This specification needs more examples and discussion of how JAX-WS’s client asynchronous model is used.

10.1 JAX-WS Client Asynchronous API for a Synchronous Service

The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client application with a means of invoking that service asynchronously, so that the client can invoke a service operation and proceed to do other work without waiting for the service operation to complete its processing. The client application can retrieve the results of the service either through a polling mechanism or via a callback method which is invoked when the operation completes.

For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006] For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100007] If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present, SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation. [JCA100008]

The additional client-side asynchronous polling and callback methods defined by JAX-WS are recognized in a Java interface as follows:

For each method M in the interface, if another method P in the interface has

- a method name that is M's method name with the characters "Async" appended, and

- the same parameter signature as M, and

- a return type of Response<R> where R is the return type of M

then P is a JAX-WS polling method that isn’t part of the SCA interface contract.

For each method M in the interface, if another method C in the interface has

- a method name that is M's method name with the characters "Async" appended, and

- a parameter signature that is M's parameter signature with an additional final parameter of type AsyncHandler<R> where R is the return type of M, and
c. a return type of Future<?>
then C is a JAX-WS callback method that isn’t part of the SCA interface contract.
As an example, an interface can be defined in WSDL as follows:

```xml
<message name="getPrice">
  <part name="ticker" type="xsd:string"/>
</message>
<message name="getPriceResponse">
  <part name="price" type="xsd:float"/>
</message>
<portType name="StockQuote">
  <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
  </operation>
</portType>
```

The JAX-WS asynchronous mapping will produce the following Java interface:

```java
// asynchronous mapping
@WebService
public interface StockQuote {
    float getPrice(String ticker);
    Response<Float> getPriceAsync(String ticker);
    Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
}
```

For SCA interface definition purposes, this is treated as equivalent to the following:

```java
// synchronous mapping
@WebService
public interface StockQuote {
    float getPrice(String ticker);
}
```

SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above example, if the client implementation uses the asynchronous form of the interface, the two additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-WS specification.
**A. XML Schema: sca-interface-java.xsd**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    elementFormDefault="qualified">
  <include schemaLocation="sca-core.xsd"/>
  <element name="interface.java" type="sca:JavaInterface"
    substitutionGroup="sca:interface"/>
  <complexType name="JavaInterface">
    <complexContent>
      <extension base="sca:Interface">
        <any namespace="##other" processContents="lax"
          minOccurs="0" maxOccurs="unbounded"/>
      </extension>
    </complexContent>
  </complexType>
</schema>
```
### B. Conformance Items

This section contains a list of conformance items for the SCA Java Common Annotations and APIs specification.

<table>
<thead>
<tr>
<th>Conformance ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCA20001</td>
<td>Remotable Services MUST NOT make use of method overloading.</td>
</tr>
<tr>
<td>JCA20002</td>
<td>The SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.</td>
</tr>
<tr>
<td>JCA20003</td>
<td>Within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.</td>
</tr>
<tr>
<td>JCA20004</td>
<td>For a composite scope implementation instance, the SCA runtime MUST ensure that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.</td>
</tr>
<tr>
<td>JCA20005</td>
<td>When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.</td>
</tr>
<tr>
<td>JCA20006</td>
<td>If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.</td>
</tr>
<tr>
<td>JCA20007</td>
<td>The SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.</td>
</tr>
<tr>
<td>JCA30001</td>
<td>The value of the @interface attribute MUST be the fully qualified name of the Java interface class.</td>
</tr>
<tr>
<td>JCA30002</td>
<td>The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks.</td>
</tr>
<tr>
<td>JCA30003</td>
<td>If the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.</td>
</tr>
<tr>
<td>JCA30004</td>
<td>The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.</td>
</tr>
<tr>
<td>JCA70001</td>
<td>SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.</td>
</tr>
</tbody>
</table>
| JCA80001       | ComponentContext.getService method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..n or 1..n.
The ComponentContext.getRequestContext method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.

When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.

An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.

SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class.

If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation.

A method annotated with @Destroy MAY have any access modifier and MUST have a void return type and no arguments.

If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends.

When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started.

A method marked with the @Init annotation MAY have any access modifier and MUST have a void return type and no arguments.

If there is a method that matches the criteria for the @Init annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.

The @Property annotation MUST NOT be used on a class field that is declared as final.

The @Property annotation MUST be used in order to inject a property onto a non-public field.

For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present.
For a @Property annotation applied to a constructor parameter, the required attribute MUST have the value true.

The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.

The @Reference annotation MUST NOT be used on a class field that is declared as final.

The @Reference annotation MUST be used in order to inject a reference onto a non-public field.

For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.

For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true.

If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..1 if the @Reference annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation required attribute is true.

If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.

An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null (either via injection or via API call).

An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection (either via injection or via API call).

References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.

In order for reinjection to occur, the following MUST be true:
1. The component MUST NOT be STATELESS scoped.
2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed.

If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed.

If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException.
If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException.

If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed.

If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference.

If the target service of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.

If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain.

If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.

If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.

In the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring.

In cases where the contents of a reference array or collection MAY change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called for any change to the contents.

A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component.
The @Remotable annotation is used to specify a Java service interface as remotable. A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.

The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.

An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.

A @Service annotation MUST only have one of the interfaces or value attribute specified.

A @Service annotation with no attributes MUST be ignored, it is the same as not having the annotation there at all.

A component implementation MUST NOT have two services with the same Java simple name.

When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes.

For a @Property annotation, if the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the implied component type of the implementation MUST have a <property/> element with a @many attribute set to true, otherwise @many is set to false.

For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.

The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.

For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.

SCA runtimes MUST support the JAXB 2.1 mapping from Java types to XML schema types.

SCA runtimes MAY support the SDO 2.1 mapping from Java types to XML schema types.

For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.

For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.
methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation. SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation.

SCA runtimes MUST support the use of the JAX-WS client asynchronous model.
The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

[Participant Name, Affiliation | Individual Member]

[Participant Name, Affiliation | Individual Member]
D. Non-Normative Text
E. Revision History

[optional; should not be included in OASIS Standards]

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2007-09-26</td>
<td>Anish Karmarkar</td>
<td>Applied the OASIS template + related changes to the Submission</td>
</tr>
<tr>
<td>2</td>
<td>2008-02-28</td>
<td>Anish Karmarkar</td>
<td>Applied resolution of issues: 4, 11, and 26</td>
</tr>
<tr>
<td>3</td>
<td>2008-04-17</td>
<td>Mike Edwards</td>
<td>Ed changes</td>
</tr>
<tr>
<td>4</td>
<td>2008-05-27</td>
<td>Anish Karmarkar</td>
<td>Added InvalidServiceException in Section 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>David Booz, Mark Combellack</td>
<td>Various editorial updates</td>
</tr>
<tr>
<td>WD04</td>
<td>2008-08-15</td>
<td>Anish Karmarkar</td>
<td>° Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) ° Applied resolution of issues 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 ° Note that issue 33 was applied, but not noted, in a previous version ° Replaced the osoa.org NS with the oasis-open.org NS</td>
</tr>
<tr>
<td>WD05</td>
<td>2008-10-03</td>
<td>Anish Karmarkar</td>
<td>° Fixed the resolution of issue 37 but re-adding the sentence: &quot;However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section ° resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references &gt; 1 ° minor ed changes</td>
</tr>
<tr>
<td>cd01-rev2</td>
<td>2008-12-12</td>
<td>Anish Karmarkar</td>
<td>° Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112</td>
</tr>
<tr>
<td>cd01-rev3</td>
<td>2008-12-16</td>
<td>David Booz</td>
<td>° Applied resolution of issues 56, 75, 111</td>
</tr>
<tr>
<td>cd02</td>
<td>2009-01-26</td>
<td>Mike Edwards</td>
<td>Minor editorial cleanup. All changes accepted.</td>
</tr>
<tr>
<td></td>
<td>Date</td>
<td>Author</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| cd02-rev1 | 2009-02-03 | Mike Edwards | Issues 25-95  
Issue 120                                                                  |
| cd02-rev2 | 2009-02-08 | Mike Edwards | Merge annotation definitions contained in section 10 into section 8  
Move remaining parts of section 10 to section 7  
Accept all changes.                                                        |
| cd02-rev3 | 2009-02-23 | Mike Edwards | RFC2119 work and formal marking of all normative statements - all sections  
Completion of Appendix B (list of all normative statements)                |
All service requests are dispatched to the same implementation instance for the lifetime of the containing composite.

A composite scoped implementation can also specify eager initialization using the @EagerInit annotation. When marked for eager initialization, the composite scoped instance is created when its containing component is started.

If a method is marked with the @Init annotation, it is called when the instance is created.

The SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and it MUST NOT perform any synchronization.

If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed.

If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw InvalidServiceException.

If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw ServiceUnavailableException.

If the target of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed.

If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is invoked on the ServiceReference.
If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the ServiceReference.

If the target of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain.

If the target has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an exception as described above.

If the target has changed, the result SHOULD be a reference to the changed service.
This means that in the cases listed above where reference reinjection is not allowed, the array or Collection for the reference MUST NOT change its contents.

In cases where the contents of a reference collection MAY change, then for references that use setter injection, the setter method MUST be called for any change to the contents.

The reinjected array or Collection MUST NOT be the same array or Collection object previously injected to the component.
<table>
<thead>
<tr>
<th>Page</th>
<th>Action</th>
<th>Username</th>
<th>Date/Time</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>Deleted</td>
<td>Mike Edwards</td>
<td>3/2/2009 2:34:00 PM</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Formatted</td>
<td>Mike Edwards</td>
<td>2/23/2009 11:11:00 AM</td>
<td>Clear (Yellow)</td>
</tr>
<tr>
<td>56</td>
<td>Formatted</td>
<td>Mike Edwards</td>
<td>2/23/2009 11:46:00 AM</td>
<td>Clear (Yellow)</td>
</tr>
</tbody>
</table>