



# Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1

Committee Specification Draft 05 /  
Public Review Draft 03

8 November 2010

**Specification URIs:**

**This Version:**

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd03.html>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd03.doc>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd03.pdf> (Authoritative)

**Previous Version:**

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.html>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.doc>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd04.pdf> (Authoritative)

**Latest Version:**

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.html>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.doc>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.pdf> (Authoritative)

**Technical Committee:**

OASIS Service Component Architecture / J (SCA-J) TC

**Chair(s):**

David Booz,	IBM
Anish Karmarkar,	Oracle Corporation

**Editor(s):**

David Booz,	IBM
Mike Edwards,	IBM
Anish Karmarkar,	Oracle Corporation

**Related work:**

This specification replaces or supersedes:

- Service Component Architecture Java Annotations and APIs Specification Version 1.00, March 21 2007

This specification is related to:

- [Service Component Architecture Assembly Model Specification Version 1.1](#)
- [SCA Policy Framework Version 1.1](#)

Compiled Java API:

<http://docs.oasis-open.org/opencsa/sca-j/sca-caa-apis-1.1-csd05.jar>

Downloadable Javadoc:

<http://docs.oasis-open.org/opencsa/sca-j/sca-j-cao-javadoc-1.1-csd05.zip>

Hosted Javadoc:

<http://docs.oasis-open.org/opencsa/sca-j/javadoc/index.html>

**Declared XML Namespace(s):**

<http://docs.oasis-open.org/ns/opencsa/sca/200912>

**Java Artifacts:**

<http://docs.oasis-open.org/opencsa/sca-j/sca-j-common-annotations-and-apis-1.1-csd05.zip>

**Abstract:**

The SCA-J Common Annotations and APIs specification defines 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 artifacts described by other SCA specifications such as [the POJO Component Implementation Specification \[JAVA\\_CI\]](#).

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 other Java-based SCA specifications can choose 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>).

**Citation Format:**

When referencing this specification the following citation format should be used:

**sca-javacaa-v1.1** OASIS Committee Specification Draft 05, *Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1*, November 2010.  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csd05.pdf>

---

## Notices

Copyright © OASIS® 2005, 2010. 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", "SCA" and "Service Component Architecture" 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 .....	7
1.1	Terminology .....	7
1.2	Normative References .....	7
1.3	Non-Normative References .....	8
2	Implementation Metadata .....	9
2.1	Service Metadata .....	9
2.1.1	@Service .....	9
2.1.2	Java Semantics of a Remotable Service .....	9
2.1.3	Java Semantics of a Local Service .....	9
2.1.4	@Reference .....	10
2.1.5	@Property .....	10
2.2	Implementation Scopes: @Scope, @Init, @Destroy .....	10
2.2.1	Stateless Scope .....	10
2.2.2	Composite Scope .....	11
2.3	@AllowsPassByReference .....	11
2.3.1	Marking Services as “allows pass by reference” .....	12
2.3.2	Marking References as “allows pass by reference” .....	12
2.3.3	Applying “allows pass by reference” to Service Proxies .....	12
2.3.4	Using “allows pass by reference” to Optimize Remotable Calls .....	13
3	Interface .....	14
3.1	Java Interface Element – <interface.java> .....	14
3.2	@Remotable .....	15
3.3	@Callback .....	15
3.4	@AsyncInvocation .....	15
3.5	SCA Java Annotations for Interface Classes .....	16
3.6	Compatibility of Java Interfaces .....	16
4	SCA Component Implementation Lifecycle .....	17
4.1	Overview of SCA Component Implementation Lifecycle .....	17
4.2	SCA Component Implementation Lifecycle State Diagram .....	17
4.2.1	Constructing State .....	18
4.2.2	Injecting State .....	18
4.2.3	Initializing State .....	19
4.2.4	Running State .....	19
4.2.5	Destroying State .....	19
4.2.6	Terminated State .....	20
5	Client API .....	21
5.1	Accessing Services from an SCA Component .....	21
5.1.1	Using the Component Context API .....	21
5.2	Accessing Services from non-SCA Component Implementations .....	21
5.2.1	SCAClientFactory Interface and Related Classes .....	21
6	Error Handling .....	23
7	Asynchronous Programming .....	24
7.1	@OneWay .....	24

7.2	Callbacks .....	24
7.2.1	Using Callbacks .....	24
7.2.2	Callback Instance Management .....	26
7.2.3	Callback Injection .....	26
7.2.4	Implementing Multiple Bidirectional Interfaces .....	26
7.2.5	Accessing Callbacks .....	27
7.3	Asynchronous handling of Long Running Service Operations.....	28
7.4	SCA Asynchronous Service Interface.....	28
8	Policy Annotations for Java .....	31
8.1	General Intent Annotations .....	31
8.2	Specific Intent Annotations .....	33
8.2.1	How to Create Specific Intent Annotations .....	34
8.3	Application of Intent Annotations .....	34
8.3.1	Intent Annotation Examples .....	35
8.3.2	Inheritance and Annotation .....	37
8.4	Relationship of Declarative and Annotated Intents .....	38
8.5	Policy Set Annotations .....	38
8.6	Security Policy Annotations .....	39
8.7	Transaction Policy Annotations .....	40
9	Java API .....	42
9.1	Component Context.....	42
9.2	Request Context.....	47
9.3	ServiceReference Interface .....	49
9.4	ResponseDispatch interface.....	50
9.5	ServiceRuntimeException .....	51
9.6	ServiceUnavailableException .....	52
9.7	InvalidServiceException .....	52
9.8	Constants .....	52
9.9	SCAClientFactory Class .....	53
9.10	SCAClientFactoryFinder Interface .....	56
9.11	SCAClientFactoryFinderImpl Class .....	57
9.12	NoSuchDomainException.....	58
9.13	NoSuchServiceException .....	58
10	Java Annotations .....	60
10.1	@AllowsPassByReference.....	60
10.2	@AsyncFault .....	61
10.3	@AsyncInvocation.....	62
10.4	@Authentication .....	62
10.5	@Authorization .....	63
10.6	@Callback .....	63
10.7	@ComponentName.....	65
10.8	@Confidentiality .....	65
10.9	@Constructor .....	66
10.10	@Context.....	67
10.11	@Destroy.....	68

10.12 @EagerInit.....	68
10.13 @Init .....	69
10.14 @Integrity .....	69
10.15 @Intent .....	70
10.16 @ManagedSharedTransaction.....	71
10.17 @ManagedTransaction .....	71
10.18 @MutualAuthentication .....	72
10.19 @NoManagedTransaction.....	73
10.20 @OneWay .....	73
10.21 @PolicySets .....	74
10.22 @Property .....	75
10.23 @Qualifier.....	76
10.24 @Reference .....	77
10.24.1 Reinjection.....	79
10.25 @Remotable.....	81
10.26 @Requires.....	83
10.27 @Scope.....	83
10.28 @Service .....	84
11 WSDL to Java and Java to WSDL.....	86
11.1 JAX-WS Annotations and SCA Interfaces .....	86
11.2 JAX-WS Client Asynchronous API for a Synchronous Service .....	92
11.3 Treatment of SCA Asynchronous Service API .....	93
12 Conformance .....	94
12.1 SCA Java XML Document .....	94
12.2 SCA Java Class.....	94
12.3 SCA Runtime.....	94
A. XML Schema: sca-interface-java-1.1.xsd.....	95
B. Java Classes and Interfaces .....	96
B.1 SCAClient Classes and Interfaces .....	96
B.1.1 SCAClientFactory Class .....	96
B.1.2 SCAClientFactoryFinder interface .....	98
B.1.3 SCAClientFactoryFinderImpl class .....	99
B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?.....	104
C. Conformance Items.....	105
D. Acknowledgements.....	121
E. Revision History .....	123

# 1 Introduction

The SCA-J Common Annotations and APIs specification defines 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 SCA Java-based 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

The goal of defining the annotations and APIs in this specification is to promote consistency and reduce duplication across the various SCA Java-based specifications. The annotations and APIs defined in this specification are designed to be used by other SCA Java-based specifications in either a partial or complete fashion.

## 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

- [RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- [ASSEMBLY] OASIS, Committee Draft 05, "06, SCA Assembly Model Specification Version 1.1", January, August 2010.  
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-ed05.pdf>  
<http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd06.pdf>
- [JAVA-CLIENT SDO] OASIS, Committee Draft 02, "SCA POJO Component Implementation Service Data Objects Specification Version 1.1", February 2010-3.0, November 2009.  
<http://www.oasis-open.org/committees/download.php/35313/sdo-3.0-cd02.zip>  
<http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-cd02.pdf>
- [SDO] SDO 2.1 Specification,  
<http://www.osea.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>
- [JAX-B] JAXB 2.1 Specification,  
<http://www.jcp.org/en/jsr/detail?id=222>
- [WSDL] WSDL Specification,  
WSDL 1.1: <http://www.w3.org/TR/wsdl>,
- [POLICY] OASIS, Committee Draft 02, "04, SCA Policy Framework Version 1.1", February 2009.  
<http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf>,  
September 2010.  
<http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd04.pdf>

45	<b>[JSR-250]</b>	Common Annotations for the Java Platform specification (JSR-250),
46		<a href="http://www.jcp.org/en/jsr/detail?id=250">http://www.jcp.org/en/jsr/detail?id=250</a>
47	<b>[JAX-WS]</b>	JAX-WS 2.1 Specification (JSR-224),
48		<a href="http://www.jcp.org/en/jsr/detail?id=224">http://www.jcp.org/en/jsr/detail?id=224</a>
49	<b>[JAVABEANS]</b>	JavaBeans 1.01 Specification,
50		<a href="http://java.sun.com/javase/technologies/desktop/javabeans/api/">http://java.sun.com/javase/technologies/desktop/javabeans/api/</a>
51	<b>[JAAS]</b>	Java Authentication and Authorization Service Reference Guide
52		<a href="http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html">http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html</a>
53		

### 54 1.3 Non-Normative References

55	<b>[EBNF-Syntax]</b>	Extended BNF syntax format used for formal grammar of constructs
56		<a href="http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation">http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation</a>
57	<b>[JAVA CI]</b>	<u>OASIS Committee Specification Draft 03, SCA POJO Component</u>
58		<u>Implementation Specification Version 1.1, November 2010.</u>
59		<a href="http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd03.pdf">http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-csd03.pdf</a>



---

## 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 provided 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 WSDL portTypes 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 or Java class that defines the service, or on a service reference. 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]

Snippet 2-1 shows an example of a Java interface for a remotable service:

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

*Snippet 2-1: Remotable Java Interface*

#### 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 or a Java class with no @Remotable annotation.

Snippet 2-2 shows an example of a Java interface for a local service:

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

*Snippet 2-2: Local Java Interface*

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 Model 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 specification 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.

Snippet 2-3 is an example showing a fragment of a service implementation annotated with lifecycle methods:

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

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

Snippet 2-3: Java Component Implementation with Lifecycle Methods

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**

146 thread at any one time. [JCA20002] In addition, within the SCA lifecycle of a stateless scoped  
147 implementation instance, the SCA runtime MUST only make a single invocation of one business method.  
148 [JCA20003] Note that the SCA lifecycle might not correspond to the Java object lifecycle due to runtime  
149 techniques such as pooling.

## 150 2.2.2 Composite Scope

151 The meaning of "composite scope" is defined in relation to the composite containing the component.  
152 It is important to distinguish between different uses of a composite, where these uses affect the numbers  
153 of instances of components within the composite. There are 2 cases:

- 154 a) Where the composite containing the component using the Java implementation is the SCA Domain  
155 (i.e. a deployment composite declares the component using the implementation)
- 156 b) Where the composite containing the component using the Java implementation is itself used as the  
157 implementation of a higher level component (any level of nesting is possible, but the component is  
158 NOT at the Domain level)

159 Where an implementation is used by a "domain level component", and the implementation is marked  
160 "Composite" scope, the SCA runtime MUST ensure that all consumers of the component appear to be  
161 interacting with a single runtime instance of the implementation. [JCA20004]

162 Where an implementation is marked "Composite" scope and it is used by a component that is nested  
163 inside a composite that is used as the implementation of a higher level component, the SCA runtime  
164 MUST ensure that all consumers of the component appear to be interacting with a single runtime instance  
165 of the implementation. There can be multiple instances of the higher level component, each running on  
166 different nodes in a distributed SCA runtime. [JCA20008]

167 The SCA runtime can exploit shared state technology in combination with other well known high  
168 availability techniques to provide the appearance of a single runtime instance for consumers of composite  
169 scoped components.

170 The lifetime of the containing composite is defined as the time it becomes active in the runtime to the time  
171 it is deactivated, either normally or abnormally.

172 When the implementation class is marked for eager initialization, the SCA runtime MUST create a  
173 composite scoped instance when its containing component is started. [JCA20005] If a method of an  
174 implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when  
175 the implementation instance is created. [JCA20006]

176 The concurrency model for the composite scope is multi-threaded. This means that the SCA runtime MAY  
177 run multiple threads in a single composite scoped implementation instance object and the SCA runtime  
178 MUST NOT perform any synchronization. [JCA20007]

## 179 2.3 @AllowsPassByReference

180 Calls to remotable services (see [section "Java Semantics of a Remotable Service"](#)) have by-value  
181 semantics. This means that input parameters passed to the service can be modified by the service  
182 without these modifications being visible to the client. Similarly, the return value or exception from the  
183 service can be modified by the client without these modifications being visible to the service  
184 implementation. For remote calls (either cross-machine or cross-process), these semantics are a  
185 consequence of marshalling input parameters, return values and exceptions "on the wire" and  
186 unmarshalling them "off the wire" which results in physical copies being made. For local method calls  
187 within the same JVM, Java language calling semantics are by-reference and therefore do not provide the  
188 correct by-value semantics for SCA remotable interfaces. To compensate for this, the SCA runtime can  
189 intervene in these calls to provide by-value semantics by making copies of any mutable objects passed.

190 The cost of such copying can be very high relative to the cost of making a local call, especially if the data  
191 being passed is large. Also, in many cases this copying is not needed if the implementation observes  
192 certain conventions for how input parameters, return values and exceptions are used. The  
193 @AllowsPassByReference annotation allows service method implementations and client references to be  
194 marked as "allows pass by reference" to indicate that they use input parameters, return values and

195 exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a  
196 remotable service is called locally within the same JVM.

### 197 **2.3.1 Marking Services as “allows pass by reference”**

198 Marking a service method implementation as “allows pass by reference” asserts that the method  
199 implementation observes the following restrictions:

- 200 • Method execution will not modify any input parameter before the method returns.
- 201 • The service implementation will not retain a reference to any mutable input parameter, mutable return  
202 value or mutable exception after the method returns.
- 203 • The method will observe “allows pass by reference” client semantics (see section 2.3.2) for any  
204 callbacks that it makes.

205 See [section “@AllowsPassByReference”](#) for details of how the @AllowsPassByReference annotation is  
206 used to mark a service method implementation as “allows pass by reference”.

### 207 **2.3.2 Marking References as “allows pass by reference”**

208 Marking a client reference as “allows pass by reference” asserts that method calls through the reference  
209 observe the following restrictions:

- 210 • The client implementation will not modify any of the method’s input parameters before the method  
211 returns. Such modifications might occur in callbacks or separate client threads.
- 212 • If the method is one-way, the client implementation will not modify any of the method’s input  
213 parameters at any time after calling the method. This is because one-way method calls return  
214 immediately without waiting for the service method to complete.

215 See [section “Applying “allows pass by reference” to Service Proxies”](#) for details of how the  
216 @AllowsPassByReference annotation is used to mark a client reference as “allows pass by reference”.

### 217 **2.3.3 Applying “allows pass by reference” to Service Proxies**

218 Service method calls are made by clients using service proxies, which can be obtained by injection into  
219 client references or by making API calls. A service proxy is marked as “allows pass by reference” if and  
220 only if any of the following applies:

- 221 • It is injected into a reference or callback reference that is marked “allows pass by reference”.
- 222 • It is obtained by calling ComponentContext.getService() or ComponentContext.getServices() with the  
223 name of a reference that is marked “allows pass by reference”.
- 224 • It is obtained by calling RequestContext.getCallback() from a service implementation that is marked  
225 “allows pass by reference”.
- 226 • It is obtained by calling ServiceReference.getService() on a service reference that is marked “allows  
227 pass by reference”.

228 A service reference for a remotable service call is marked “allows pass by reference” if and only if any of  
229 the following applies:

- 230 • It is injected into a reference or callback reference that is marked “allows pass by reference”.
- 231 • It is obtained by calling ComponentContext.getServiceReference() or  
232 ComponentContext.getServiceReferences() with the name of a reference that is marked “allows pass  
233 by reference”.
- 234 • It is obtained by calling RequestContext.getCallbackReference() from a service implementation that is  
235 marked “allows pass by reference”.
- 236 • It is obtained by calling ComponentContext.cast() on a proxy that is marked “allows pass by  
237 reference”.

#### 238     **2.3.4 Using “allows pass by reference” to Optimize Remotable Calls**

239     The SCA runtime MAY use by-reference semantics when passing input parameters, return values or  
240     exceptions on calls to remotable services within the same JVM if both the service method implementation  
241     and the service proxy used by the client are marked “allows pass by reference”. [JCA20009]

242     The SCA runtime MUST use by-value semantics when passing input parameters, return values and  
243     exceptions on calls to remotable services within the same JVM if the service method implementation is  
244     not marked “allows pass by reference” or the service proxy used by the client is not marked “allows pass  
245     by reference”. [JCA20010]

## 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 SCA Documents 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.

It is possible that the Java interface class referenced by the <interface.java/> element contains one or more annotations defined by the JAX-WS specification [JAX-WS]. These annotations can affect the interpretation of the <interface.java/> element. In the most extreme case, the annotations cause the replacement of the <interface.java/> element with an <interface.wsdl/> element. The relevant JAX-WS annotations and their effects on the <interface.java/> element are described in the section "JAX-WS Annotations and SCA Interfaces".

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

Snippet 3-1 is the pseudo-schema for the interface.java element

```
<interface.java interface="NCName" callbackInterface="NCName"?  
    requires="list of xs:QName"?  
    policySets="list of xs:QName"?  
    remotable="boolean"?/>
```

Snippet 3-1: interface.java Pseudo-Schema

The interface.java element has the 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]  
If the identified class is annotated with either the JAX-WS @WebService or @WebServiceProvider annotations and the annotation has a non-empty wsdlLocation property, then the SCA Runtime MUST act as if an <interface.wsdl/> element is present instead of the <interface.java/> element, with an @interface attribute identifying the portType mapped from the Java interface class and containing @requires and @policySets attribute values equal to the @requires and @policySets attribute values of the <interface.java/> element. [JCA30010]
- **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]
- **requires : QName (0..1)** – a list of policy intents. See the Policy Framework specification [POLICY] for a description of this attribute
- **policySets : QName (0..1)** – a list of policy sets. See the Policy Framework specification [POLICY] for a description of this attribute.
- **remotable : boolean (0..1)** – indicates whether or not the interface is remotable. A value of "true" means the interface is remotable and a value of "false" means it is not. This attribute does not have a default value. If it is not specified then the remotability is determined by the presence or absence of the @Remotable annotation on the interface class. The @remotable attribute applies to both the interface and any optional callbackInterface. The @remotable attribute is intended as an alternative to using the @Remotable annotation on the interface class. The value of the @remotable attribute

on the <interface.java/> element does not override the presence of a @Remotable annotation on the interface class and so if the interface class contains a @Remotable annotation and the @remotable attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the component concerned. [JCA30005]

Snippet 3-2 shows an example of the Java interface element:

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

Snippet 3-2 Example interface.java Element

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, a service implementation class, or a service reference denotes an interface or class that is designed to be used for remote communication. Remotable interfaces are intended to be used for **coarse grained** services. Operations' parameters, return values and exceptions 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.

## 3.4 @AsyncInvocation

An interface can be annotated with @AsyncInvocation or with the equivalent @Requires("sca:asyncInvocation") annotation to indicate that request/response operations of that interface are **long running** and that response messages are likely to be sent an arbitrary length of time after the initial request message is sent to the target service. This is described in the [SCA Assembly Specification \[ASSEMBLY\]](#).

For a service client, it is strongly recommended that the client uses the asynchronous form of the client interface when using a reference to a service with an interface annotated with @AsyncInvocation, using either polling or callbacks to receive the response message. See the sections "[Asynchronous Programming](#)" and the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)" for more details about the asynchronous client API.

337 For a service implementation, SCA provides an **asynchronous service** mapping of the WSDL  
338 request/response interface which enables the service implementation to send the response message at  
339 an arbitrary time after the original service operation is invoked. This is described in the section  
340 "Asynchronous handling of Long Running Service Operations".

### 341 3.5 SCA Java Annotations for Interface Classes

342 A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT  
343 contain any of the following SCA Java annotations:

344 @AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit,  
345 @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30006]

346 A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST  
347 NOT contain any of the following SCA Java annotations:

348 @AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy,  
349 @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service. [JCA30007]

### 350 3.6 Compatibility of Java Interfaces

351 The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be  
352 satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship.  
353 If these interfaces are both Java interfaces, compatibility also means that every method that is present in  
354 both interfaces is defined consistently in both interfaces with respect to the @OneWay annotation, that is,  
355 the annotation is either present in both interfaces or absent in both interfaces. [JCA30009]



---

## 4 SCA Component Implementation Lifecycle

This section describes the lifecycle of an SCA component implementation.

### 4.1 Overview of SCA Component Implementation Lifecycle

At a high level, there are 3 main phases through which an SCA component implementation will transition when it is used by an SCA Runtime:

- **The Initialization phase.** This involves constructing an instance of the component implementation class and injecting any properties and references. Once injection is complete, the method annotated with `@Init` is called, if present, which provides the component implementation an opportunity to perform any internal initialization it requires.
- **The Running phase.** This is where the component implementation has been initialized and the SCA Runtime can dispatch service requests to it over its Service interfaces.
- **The Destroying phase.** This is where the component implementation's scope has ended and the SCA Runtime destroys the component implementation instance. The SCA Runtime calls the method annotated with `@Destroy`, if present, which provides the component implementation an opportunity to perform any internal clean up that is required.

### 4.2 SCA Component Implementation Lifecycle State Diagram

The state diagram in Figure 4-1 shows the lifecycle of an SCA component implementation. The sections that follow it describe each of the states that it contains.

It should be noted that some component implementation specifications might not implement all states of the lifecycle. In this case, that state of the lifecycle is skipped over.

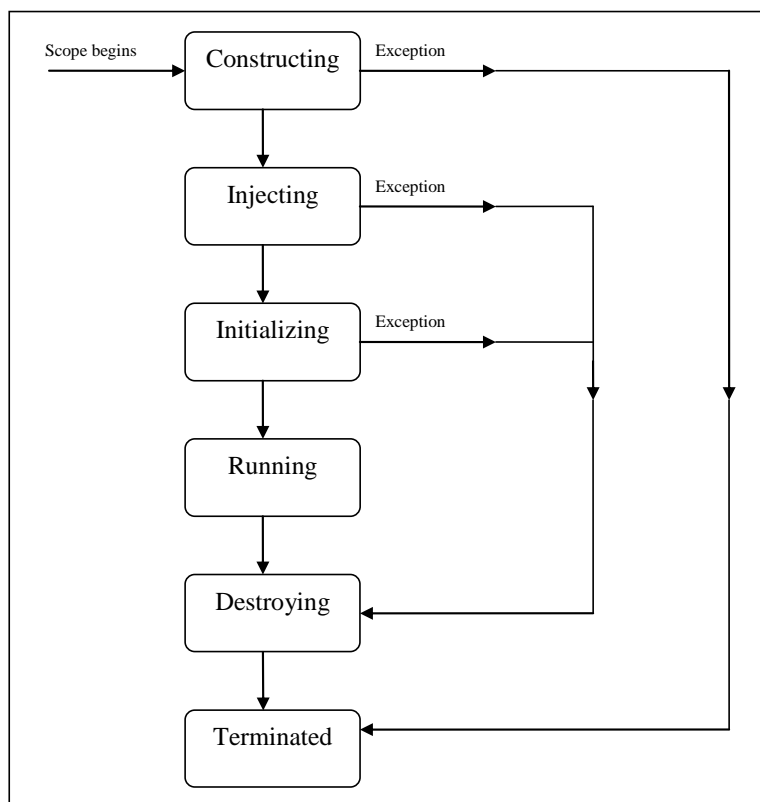


Figure 4-1: SCA - Component Implementation Lifecycle

## 4.2.1 Constructing State

The SCA Runtime MUST call a constructor of the component implementation at the start of the Constructing state. [JCA40001] The SCA Runtime MUST perform any constructor reference or property injection when it calls the constructor of a component implementation. [JCA40002]

The result of invoking operations on any injected references when the component implementation is in the Constructing state is undefined.

When the constructor completes successfully, the SCA Runtime MUST transition the component implementation to the Injecting state. [JCA40003] If an exception is thrown whilst in the Constructing state, the SCA Runtime MUST transition the component implementation to the Terminated state. [JCA40004]

## 4.2.2 Injecting State

When a component implementation instance is in the Injecting state, the SCA Runtime MUST first inject all field and setter properties that are present into the component implementation. [JCA40005] The order in which the properties are injected is unspecified.

When a component implementation instance is in the Injecting state, the SCA Runtime MUST inject all field and setter references that are present into the component implementation, after all the properties have been injected. [JCA40006] The order in which the references are injected is unspecified.

395 The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected  
396 properties and references are made visible to the component implementation without requiring the  
397 component implementation developer to do any specific synchronization. [JCA40007]  
398 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the  
399 component implementation is in the Injecting state. [JCA40008]  
400 The result of invoking operations on any injected references when the component implementation is in  
401 the Injecting state is undefined.  
402 When the injection of properties and references completes successfully, the SCA Runtime MUST  
403 transition the component implementation to the Initializing state. [JCA40009] If an exception is thrown  
404 whilst injecting properties or references, the SCA Runtime MUST transition the component  
405 implementation to the Destroying state. [JCA40010] If a property or reference is unable to be injected, the  
406 SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40024]

#### 407 4.2.3 Initializing State

408 When the component implementation enters the Initializing State, the SCA Runtime MUST call the  
409 method annotated with @Init on the component implementation, if present. [JCA40011]  
410 The component implementation can invoke operations on any injected references when it is in the  
411 Initializing state. However, depending on the order in which the component implementations are  
412 initialized, the target of the injected reference might not be available since it has not yet been initialized. If  
413 a component implementation invokes an operation on an injected reference that refers to a target that has  
414 not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException. [JCA40012]  
415 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the  
416 component implementation instance is in the Initializing state. [JCA40013]  
417 Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the  
418 component implementation to the Running state. [JCA40014] If an exception is thrown whilst initializing,  
419 the SCA Runtime MUST transition the component implementation to the Destroying state. [JCA40015]

#### 420 4.2.4 Running State

421 The SCA Runtime MUST invoke Service methods on a component implementation instance when the  
422 component implementation is in the Running state and a client invokes operations on a service offered by  
423 the component. [JCA40016]  
424 The component implementation can invoke operations on any injected references when the component  
425 implementation instance is in the Running state.  
426 When the component implementation scope ends, the SCA Runtime MUST transition the component  
427 implementation to the Destroying state. [JCA40017]

#### 428 4.2.5 Destroying State

429 When a component implementation enters the Destroying state, the SCA Runtime MUST call the method  
430 annotated with @Destroy on the component implementation, if present. [JCA40018]  
431 The component implementation can invoke operations on any injected references when it is in the  
432 Destroying state. However, depending on the order in which the component implementations are  
433 destroyed, the target of the injected reference might no longer be available since it has been destroyed. If  
434 a component implementation invokes an operation on an injected reference that refers to a target that has  
435 been destroyed, the SCA Runtime MUST throw an InvalidServiceException. [JCA40019]  
436 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the  
437 component implementation instance is in the Destroying state. [JCA40020]  
438 Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST transition  
439 the component implementation to the Terminated state. [JCA40021] If an exception is thrown whilst  
440 destroying, the SCA Runtime MUST transition the component implementation to the Terminated state.  
441 [JCA40022]

#### 442     **4.2.6 Terminated State**

443     The lifecycle of the SCA Component has ended.

444     The SCA Runtime MUST NOT invoke Service methods on the component implementation when the  
445     component implementation instance is in the Terminated state. [JCA40023]

---

## 5 Client API

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

### 5.1 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.

#### 5.1.1 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.

### 5.2 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.

#### 5.2.1 SCAClientFactory Interface and Related Classes

Client code can use the **SCAClientFactory** class to obtain proxy reference objects for a service which is in an SCA Domain. The URI of the domain, the relative URI of the service and the business interface of the service must all be known in order to use the SCAClientFactory class.

Objects which implement the SCAClientFactory are obtained using the newInstance() methods of the SCAClientFactory class.

Snippet 5-1 is a sample of the code that a client would use:

```
package org.oasisopen.sca.client.example;

import java.net.URI;

import org.oasisopen.sca.client.SCAClientFactory;
import org.oasisopen.sca.client.example.HelloService;

/**
 * Example of use of Client API for a client application to obtain
 * an SCA reference proxy for a service in an SCA Domain.
 */
public class Client1 {

    public void someMethod() {

        try {

            String serviceURI = "SomeHelloServiceURI";
            URI domainURI = new URI("SomeDomainURI");

            SCAClientFactory scaClient =
                SCAClientFactory.newInstance( domainURI );
            HelloService helloService =
                scaClient.getService(HelloService.class,
                                    serviceURI);
```

```
494         String reply = helloService.sayHello("Mark");
495
496     } catch (Exception e) {
497         System.out.println("Received exception");
498     }
499 }
500 }
```

501 *Snippet 5-1: Using the SCAClientFactory Interface*

502

503 For details about the SCAClientFactory interface and its related classes see the section  
504 ["SCAClientFactory Class"](#).

---

## 505 6 Error Handling

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

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

509 SCA runtime exceptions are raised by the SCA runtime and signal problems in management of  
510 component execution or problems interacting with remote services. The SCA runtime exceptions are  
511 defined in [the Java API section](#).

---

## 7 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.

### 7.1 @OneWay

**Non-blocking 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.

A 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 throw exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in [the section "JAX-WS Client Asynchronous API for a Synchronous Service"](#). 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.

### 7.2 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 Model specification \[ASSEMBLY\]](#).

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.

#### 7.2.1 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.

Snippet 7-1 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.



```

556
557 package somepackage;
558 import org.oasisopen.sca.annotation.Callback;
559 import org.oasisopen.sca.annotation.Remotable;
560
561 @Remotable
562 @Callback(QuotationCallback.class)
563 public interface Quotation {
564     double requestQuotation(String productCode, int quantity);
565 }
566
567 @Remotable
568 public interface QuotationCallback {
569     String getState();
570     String getZipCode();
571     String getCreditRating();
572 }

```

Snippet 7-1: Using a Bidirectional Interface

In Snippet 7-1, 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. Snippet 7-2 illustrates a possible implementation of the example service, using the `@Callback` annotation to request that a callback proxy be injected.

```

584 @Callback
585 protected QuotationCallback callback;
586
587 public double requestQuotation(String productCode, int quantity) {
588     double price = getPrice(productCode, quantity);
589     double discount = 0;
590     if (quantity > 1000 && callback.getState().equals("FL")) {
591         discount = 0.05;
592     }
593     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
594         discount += 0.05;
595     }
596     return price * (1-discount);
597 }

```

Snippet 7-2: Example Implementation of a Service with a Bidirectional Interface

Snippet 7-3 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`.

```

603
604 public class ClientImpl implements ClientService, QuotationCallback {
605
606     private QuotationService myService;
607
608     @Reference
609     public void setMyService(QuotationService service) {
610         myService = service;
611     }
612 }

```

```

613 public void aClientMethod() {
614     ...
615     double quote = myService.requestQuotation("AB123", 2000);
616     ...
617 }
618
619 public String getState() {
620     return "TX";
621 }
622 public String getZipCode() {
623     return "78746";
624 }
625 public String getCreditRating() {
626     return "AA";
627 }
628 }

```

Snippet 7-3: Example Client Using a Bidirectional Interface

Snippet 7-3 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.

## 7.2.2 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 the section "Using Callbacks", 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.

## 7.2.3 Callback Injection

When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the invoking service into all fields and setter methods of the service implementation class that are marked with a @Callback annotation and typed by the callback interface of the bidirectional service, and the SCA runtime MUST inject null into all other fields and setter methods of the service implementation class that are marked with a @Callback annotation. [JCA60001] When a non-bidirectional service is invoked, the SCA runtime MUST inject null into all fields and setter methods of the service implementation class that are marked with a @Callback annotation. [JCA60002]

## 7.2.4 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. Snippet 7-4 shows the declaration of two fields, each of which corresponds to a particular service offered by the implementation.

```

662 @Callback
663 protected MyService1Callback callback1;

```

```
@Callback
protected MyService2Callback callback2;
```

*Snippet 7-4: Multiple Bidirectional Interfaces in an Implementation*

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

## 7.2.5 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()`.

Snippet 7-5 comes from a service implementation that uses the callback API:

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

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

*Snippet 7-5: Using the Callback API*

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. Snippet 7-6 shows how to retrieve a callback in a method programmatically:

```
@Context
ComponentContext context;

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

*Snippet 7-6: Using RequestContext to get a Callback*

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

## 7.3 Asynchronous handling of Long Running Service Operations

Long-running request-response operations are described in the SCA Assembly Specification [ASSEMBLY]. These operations are characterized by following the WSDL request-response message exchange pattern, but where the timing of the sending of the response message is arbitrarily later than the receipt of the request message, with an impact on the client component, on the service component and also on the transport binding used to communicate between them.

In SCA, such operations are marked with an intent "asyncInvocation" and is expected that the client component, the service component and the binding are all affected by the presence of this intent. This specification does not describe the effects of the intent on the binding, other than to note that in general, there is an implication that the sending of the response message is typically separate from the sending of the request message, typically requiring a separate response endpoint on the client to which the response can be sent.

For components that are clients of a long-running request-response operation, it is strongly recommended that the client makes use of the JAX-WS Client Asynchronous API, either using the polling interface or the callback mechanism described in the section "[JAX-WS Client Asynchronous API for a Synchronous Service](#)". The principle is that the client should not synchronously wait for a response from the long running operation since this could take a long time and it is preferable not to tie up resources while waiting.

For the service implementation component, the JAX-WS client asynchronous API is not suitable, so the SCA Java Common Annotations and APIs specification defines the SCA Asynchronous Service interface, which, like the JAX-WS client asynchronous API, is an alternative mapping of a WSDL request-response operation into a Java interface.

## 7.4 SCA Asynchronous Service Interface

The SCA Asynchronous Service interface follows some of the patterns defined by the JAX-WS client asynchronous API, but it is a simpler interface aligned with the needs of a service implementation class.

As an example, for a WSDL portType with a single operation "getPrice" with a String request parameter and a float response, the synchronous Java interface mapping appears in Snippet 7-7.

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

*Snippet 7-7: Example Synchronous Java Interface Mapping*

The JAX-WS client asynchronous API for the same portType adds two asynchronous forms for each synchronous method, as shown in Snippet 7-8.

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

*Snippet 7-8: Example JAX-WS Client Asynchronous Java interface Mapping*

The SCA Asynchronous Service interface has a single method similar to the final one in the asynchronous client interface, as shown in Snippet 7-8.

```
// asynchronous mapping
```

```

761 @Requires("sca:asyncInvocation")
762 public interface StockQuote {
763     void getPriceAsync(String ticker, ResponseDispatch<Float> dispatch);
764 }

```

765 *Snippet 7-9: Example SCA Asynchronous Service Java interface Mapping*

766

767 The main characteristics of the SCA asynchronous mapping are:

- 768 • there is a single method, with a name with the string "Async" appended to the operation name
- 769 • it has a void return type
- 770 • it has two input parameters, the first is the request message of the operation and the second is a  
771 ResponseDispatch object typed by the response message of the operation (following the rules  
772 expressed in the JAX-WS specification for the typing of the AsyncHandler object in the client  
773 asynchronous API)
- 774 • it is annotated with the asyncInvocation intent
- 775 • if the synchronous method has any business faults/exceptions, it is annotated with @AsyncFault,  
776 containing a list of the exception classes

777 Unlike the JAX-WS asynchronous client interface, there is only a single operation for the service  
778 implementation to provide (it would be inconvenient for the service implementation to be required to  
779 implement multiple methods for each operation in the WSDL interface).

780 The ResponseDispatch parameter is the mechanism by which the service implementation sends back the  
781 response message resulting from the invocation of the service method. The ResponseDispatch is  
782 serializable and it can be invoked once at any time after the invocation of the service method, either  
783 before or after the service method returns. This enables the service implementation to store the  
784 ResponseDispatch in serialized form and release resources while waiting for the completion of whatever  
785 activities result from the processing of the initial invocation.

786 The ResponseDispatch object is allocated by the SCA runtime/binding implementation and it is expected  
787 to contain whatever metadata is required to deliver the response message back to the client that invoked  
788 the service operation.

789 The SCA asynchronous service Java interface mapping of a WSDL request-response operation  
790 MUST appear as follows:

791 The interface is annotated with the "asyncInvocation" intent.

792 – For each service operation in the WSDL, the Java interface contains an operation with

793 – a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async"  
794 added

795 – a void return type

796 – a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the  
797 WSDL operation plus an additional last parameter which is a ResponseDispatch object typed by  
798 the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where  
799 ResponseDispatch is the type defined in the SCA Java Common Annotations and APIs  
800 specification. [JCA60003]

801 An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an  
802 SCA service. [JCA60004]

803 The ResponseDispatch object passed in as a parameter to a method of a service implementation using  
804 the SCA asynchronous service Java interface can be invoked once only through either its sendResponse  
805 method or through its sendFault method to return the response resulting from the service method  
806 invocation. If the SCA asynchronous service interface ResponseDispatch handleResponse method is  
807 invoked more than once through either its sendResponse or its sendFault method, the SCA runtime  
808 MUST throw an IllegalStateException. [JCA60005]

809

810 For the purposes of matching interfaces (when wiring between a reference and a service, or when using  
811 an implementation class by a component), an interface which has one or more methods which follow the  
812 SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent  
813 synchronous methods, as follows:

814 Asynchronous service methods are characterized by:

- 815 – void return type
- 816 – a method name with the suffix "Async"
- 817 – a last input parameter with a type of ResponseDispatch<X>
- 818 – annotation with the asyncInvocation intent
- 819 – possible annotation with the @AsyncFault annotation

820 The mapping of each such method is as if the method had the return type "X", the method name without  
821 the suffix "Async" and all the input parameters except the last parameter of the type  
822 ResponseDispatch<X>, plus the list of exceptions contained in the @AsyncFault annotation. [JCA60006]

## 8 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 expects 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 and Transactions.

This specification supports using [the Common Annotations for the 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. SCA policy annotation semantics follow the General Guidelines for Inheritance of Annotations in [the Common Annotations for the Java Platform specification \[JSR-250\]](#), except that member-level annotations in a class or interface do not have any effect on how class-level annotations are applied to other members of the class or interface.

### 8.1 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 shown in Snippet 8-1.

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

*Snippet 8-1: Intent Format*

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 those in Snippet 8-2.

```

869 public static final String SCA_PREFIX =
870     "{http://docs.oasis-open.org/ns/opencsa/sca/200912}";
871 public static final String CONFIDENTIALITY =
872     SCA_PREFIX + "confidentiality";
873 public static final String CONFIDENTIALITY_MESSAGE =
874     CONFIDENTIALITY + ".message";

```

875 *Snippet 8-2: Example Intent Constants*

876

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

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

882 An example of the `@Requires` annotation with 2 qualified intents (from the Security domain) is shown in  
883 Snippet 8-3:

884

```

885 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})

```

886 *Snippet 8-3: Multiple Intents in One Annotation*

887

888 The annotation in Snippet 8-3 attaches the intents "confidentiality.message" and "integrity.message".

889 Snippet 8-4 is an example of a reference requiring support for confidentiality:

890

```

891 package com.foo;
892
893 import static org.oasisopen.sca.annotation.Confidentiality.*;
894 import static org.oasisopen.sca.annotation.Reference;
895 import static org.oasisopen.sca.annotation.Requires;
896
897 public class Foo {
898     @Requires(CONFIDENTIALITY)
899     @Reference
900     public void setBar(Bar bar) {
901         ...
902     }
903 }

```

904 *Snippet 8-4: Annotation a Reference*

905

906 Users can also choose to only use constants for the namespace part of the QName, so that they can add  
907 new intents without having to define new constants. In that case, the definition of Snippet 8-4 would  
908 instead look like Snippet 8-5.

909

```

910 package com.foo;
911
912 import static org.oasisopen.sca.Constants.*;
913 import static org.oasisopen.sca.annotation.Reference;
914 import static org.oasisopen.sca.annotation.Requires;
915
916 public class Foo {
917     @Requires(SCA_PREFIX+"confidentiality")
918     @Reference
919     public void setBar(Bar bar) {
920         ...
921     }
922 }

```



Snippet 8-5: Using Intent Constants and strings

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

```
'@Requires('' QualifiedIntent ''' (''' QualifiedIntent ''')* '')
```

where

```
QualifiedIntent ::= QName('.' Qualifier)*  
Qualifier ::= NCName
```

See [section @Requires](#) for the formal definition of the @Requires annotation.

## 8.2 Specific Intent Annotations

In addition to the general intent annotation supplied by the @Requires annotation described in section 8.2, 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 shown in Snippet 8-6.

```
@Integrity
```

Snippet 8-6: Example Specific Intent Annotation

An example of a qualified specific intent for the "authentication" intent is shown in Snippet 8-7.

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

Snippet 8-7: Example Qualified Specific Intent Annotation

This annotation attaches the pair of qualified intents: "authentication.message" and "authentication.transport" (the sca: namespace is assumed in this both of these cases – "http://docs.oasis-open.org/ns/opencsa/sca/200912").

The general form of specific intent annotations is shown in Snippet 8-8

```
'@' Intent ('(' qualifiers ')')?
```

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

```
Intent ::= NCName  
qualifiers ::= ''' qualifier ''' (''' qualifier ''')*  
qualifier ::= NCName ('.' qualifier)?
```

Snippet 8-8: Specific Intent Annotation Format

## 8.2.1 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. [JCA70001]

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 (although not required) 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, as shown in Snippet 8-9:

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

*Snippet 8-9: Defining a Specific Intent Annotation*

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 the annotation in Snippet 8-10

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

*Snippet 8-10: Multiple Qualifiers in an Annotation*

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](#).

## 8.3 Application of Intent Annotations

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

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

**Intent annotations MUST NOT be applied to the following:**

- A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70002]

Intent annotations can be applied to classes, interfaces, and interface methods. Applying an intent annotation to a field, setter method, or constructor parameter allows intents to be defined at references. Intent annotations can also be applied to reference interfaces and their methods.

Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA runtime MUST compute the combined intents for the Java element by merging the intents from all intent annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging intents at the same hierarchy level. [JCA70003]

An example of multiple policy annotations being used together is shown in Snippet 8-11:

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

*Snippet 8-11: Multiple Policy Annotations*

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

If intent annotations are specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective intents for the method by merging the combined intents from the method with the combined intents for the interface according to the SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the method at the lower level and the interface at the higher level. [JCA70004] This merging process does not remove or change any intents that are applied to the interface.

### 8.3.1 Intent Annotation Examples

The following examples show how the rules defined in section 8.3 are applied.

Snippet 8-12 shows how intents on references are merged. In this example, the intents for `myRef` are "authentication" and "confidentiality.message".

```
@Authentication
@Requires(CONFIDENTIALITY)
@Confidentiality("message")
@Reference
protected MyService myRef;
```

*Snippet 8-12: Merging Intents on References*

Snippet 8-13 shows that mutually exclusive intents cannot be applied to the same Java element. In this example, the Java code is in error because of contradictory mutually exclusive intents "managedTransaction" and "noManagedTransaction".

```
@Requires({SCA_PREFIX+"managedTransaction",
           SCA_PREFIX+"noManagedTransaction"})
@Reference
protected MyService myRef;
```

*Snippet 8-13: Mutually Exclusive Intents*

Snippet 8-14 shows that intents can be applied to Java service interfaces and their methods. In this example, the effective intents for `MyService.mymethod()` are "authentication" and "confidentiality".

```
@Authentication
public interface MyService {
```

```

1055     @Confidentiality
1056     public void mymethod();
1057 }
1058 @Service(MyService.class)
1059 public class MyServiceImpl {
1060     public void mymethod() {...}
1061 }

```

Snippet 8-14: *Intents on Java Interfaces, Interface Methods, and Java Classes*

Snippet 8-15 shows that intents can be applied to Java service implementation classes. In this example, the effective intents for `MyService.mymethod()` are "authentication", "confidentiality", and "managedTransaction".

```

1068 @Authentication
1069 public interface MyService {
1070     @Confidentiality
1071     public void mymethod();
1072 }
1073 @Service(MyService.class)
1074 @Requires(SCA_PREFIX+"managedTransaction")
1075 public class MyServiceImpl {
1076     public void mymethod() {...}
1077 }

```

Snippet 8-15: *Intents on Java Service Implementation Classes*

Snippet 8-16 shows that intents can be applied to Java reference interfaces and their methods, and also to Java references. In this example, the effective intents for the method `mymethod()` of the reference `myRef` are "authentication", "integrity", and "confidentiality".

```

1084 @Authentication
1085 public interface MyRefInt {
1086     @Integrity
1087     public void mymethod();
1088 }
1089 @Service(MyService.class)
1090 public class MyServiceImpl {
1091     @Confidentiality
1092     @Reference
1093     protected MyRefInt myRef;
1094 }

```

Snippet 8-16: *Intents on Java References and their Interfaces and Methods*

Snippet 8-17 shows that intents cannot be applied to methods of Java implementation classes. In this example, the Java code is in error because of the `@Authentication` intent annotation on the implementation method `MyServiceImpl.mymethod()`.

```

1101 public interface MyService {
1102     public void mymethod();
1103 }
1104 @Service(MyService.class)
1105 public class MyServiceImpl {
1106     @Authentication
1107     public void mymethod() {...}
1108 }

```

*Snippet 8-17: Intent on Implementation Method*

Snippet 8-18 shows one effect of applying the SCA Policy Framework rules for merging intents within a structural hierarchy to Java service interfaces and their methods. In this example a qualified intent overrides an unqualified intent, so the effective intent for `MyService.mymethod()` is "confidentiality.message".

```
@Confidentiality("message")
public interface MyService {
    @Confidentiality
    public void mymethod();
}
```

*Snippet 8-18: Merging Qualified and Unqualified Intents on Java Interfaces and Methods*

Snippet 8-19 shows another effect of applying the SCA Policy Framework rules for merging intents within a structural hierarchy to Java service interfaces and their methods. In this example a lower-level intent causes a mutually exclusive higher-level intent to be ignored, so the effective intent for `mymethod1()` is "managedTransaction" and the effective intent for `mymethod2()` is "noManagedTransaction".

```
@Requires(SCA_PREFIX+"managedTransaction")
public interface MyService {
    public void mymethod1();
    @Requires(SCA_PREFIX+"noManagedTransaction")
    public void mymethod2();
}
```

*Snippet 8-19: Merging Mutually Exclusive Intents on Java Interfaces and Methods*

### 8.3.2 Inheritance and Annotation

Snippet 8-20 shows the inheritance relations of intents on classes, operations, and super classes.

```
package services.hello;
import org.oasisopen.sca.annotation.Authentication;
import org.oasisopen.sca.annotation.Integrity;

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

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

package services.hello;
import org.oasisopen.sca.annotation.Authentication;
import org.oasisopen.sca.annotation.Confidentiality;

@Confidentiality("message")
public class HelloChildService extends HelloService {
    @Confidentiality("transport")
    public String hello(String message) {...}
    @Authentication
    String helloWorld() {...}
}
```

Snippet 8-20: Usage example of Annotated Policy and Inheritance

The effective intent annotation on the **helloWorld** method of **HelloChildService** is @Authentication and @Confidentiality("message").  
The effective intent annotation on the **hello** method of **HelloChildService** is @Confidentiality("transport"),  
The effective intent annotation on the **helloThere** method of **HelloChildService** is @Integrity and @Authentication("transport"), the same as for this method in the **HelloService** class.  
The effective intent annotation on the **hello** method of **HelloService** is @Integrity and @Authentication("message")

Table 8-1 shows the equivalent declarative security interaction policy of the methods of the HelloService and HelloChildService implementations corresponding to the Java classes shown in Snippet 8-20.

Class	Method		
	hello()	helloThere()	helloWorld()
HelloService	integrity authentication.message	integrity authentication.transport	N/A
HelloChildService	confidentiality.transport	integrity authentication.transport	authentication confidentiality.message

Table 8-1: Declarative Intents Equivalent to Annotated Intents in Snippet 8-20

## 8.4 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.

## 8.5 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({' policySetQName (' policySetQName )* '})'
```

Snippet 8-21: PolicySet Annotation Format

As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".  
An example of the @PolicySets annotation is shown in Snippet 8-22:

```
@Reference(name="helloService", required=true)
```

```

@PolicySets({ MY_NS + "WS Encryption Policy",
              MY_NS + "WS Authentication Policy" })
public setHelloService(HelloService service) {
    . . .
}

```

*Snippet 8-22: Use of @PolicySets*

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

The `@PolicySets` annotation MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated with `@Reference` or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with `@Reference`

[JCA70005]

The `@PolicySets` annotation can be applied to classes, interfaces, and interface methods. Applying a `@PolicySets` annotation to a field, setter method, or constructor parameter allows policy sets to be defined at references. The `@PolicySets` annotation can also be applied to reference interfaces and their methods.

If the `@PolicySets` annotation is specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy sets from the method with the policy sets from the interface. [JCA70006] This merging process does not remove or change any policy sets that are applied to the interface.

## 8.6 Security Policy Annotations

This section introduces annotations for commonly used SCA security intents, as defined in [the SCA Policy Framework Specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for additional security policy intents that can be used with the `@Requires` annotation. The following annotations for security policy intents and qualifiers are defined:

- `@Authentication`
- `@Authorization`
- `@Confidentiality`
- `@Integrity`
- `@MutualAuthentication`

The `@Authentication`, `@Confidentiality`, and `@Integrity` intents have the same pair of Qualifiers:

- `message`
- `transport`

The formal definitions of the security intent annotations are found in the section “Java Annotations”.

Snippet 8-23 shows an example of applying security intents 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.

```
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;

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

@Service(ClientService.class)
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!");
        ...
    }
}
```

Snippet 8-23: Usage of Security Intents on a Reference

## 8.7 Transaction Policy Annotations

This section introduces annotations for commonly used SCA transaction intents, as defined in [the SCA Policy Framework specification \[POLICY\]](#). Also see the SCA Policy Framework Specification for additional transaction policy intents that can be used with the @Requires annotation. The following annotations for transaction policy intents and qualifiers are defined:

- @ManagedTransaction
- @NoManagedTransaction
- @SharedManagedTransaction

The @ManagedTransaction intent has the following Qualifiers:

- global
- local

The formal definitions of the transaction intent annotations are found in the section “Java Annotations”.



1299 Snippet 8-24 shows an example of applying a transaction intent to a component implementation, where  
1300 the component implementation requires a global transaction.

1301

```
1302 package services.hello;  
1303 // Interface for HelloService  
1304 public interface HelloService {  
1305     String hello(String helloMsg);  
1306 }  
1307  
1308 // Implementation class for HelloService  
1309 package services.hello.impl;  
1310  
1311 import services.hello.HelloService;  
1312 import org.oasisopen.sca.annotation.*;  
1313  
1314 @Service(HelloService.class)  
1315 @ManagedTransaction("global")  
1316 public class HelloServiceImpl implements HelloService {  
1317  
1318     public void someMethod() {  
1319         ...  
1320     }  
1321 }
```

1322 *Snippet 8-24: Usage of Transaction Intents in an Implementation*

## 9 Java API

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

### 9.1 Component Context

Figure 9-1 defines the **ComponentContext** interface:

```
package org.oasisopen.sca;
import java.util.Collection;
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);

    <B> 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);

    RequestContext getRequestContext();

    <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;

}
```

Figure 9-1: ComponentContext Interface

#### **getURI () method:**

Returns the **absolute structural** URI [ASSEMBLY] of the component within the SCA Domain.

Returns:

- **String** which contains the absolute URI of the component in the SCA Domain  
The ComponentContext.getURI method MUST return the **structural** URI of the component in the SCA Domain. [JCA80008]

Parameters:

- **none**

Exceptions:

- **none**

#### **getService ( Class<B> businessInterface, String referenceName ) method:**

1371 Returns a typed service proxy object for a reference defined by the current component, where the  
1372 reference has multiplicity 0..1 or 1..1.

1373 Returns:

- 1374 • **B** which is a proxy object for the reference, which implements the interface B contained in the  
1375 businessInterface parameter.

1376 The ComponentContext.getService method MUST return the proxy object implementing the interface  
1377 provided by the businessInterface parameter, for the reference named by the referenceName  
1378 parameter with the interface defined by the businessInterface parameter when that reference has a  
1379 target service configured. [JCA80009]

1380 The ComponentContext.getService method MUST return null if the multiplicity of the reference  
1381 named by the referenceName parameter is 0..1 and the reference has no target service configured.  
1382 [JCA80010]

1383 Parameters:

- 1384 • **Class<B> businessInterface** - the Java interface for the service reference
- 1385 • **String referenceName** - the name of the service reference

1386 Exceptions:

- 1387 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the  
1388 reference identified by the referenceName parameter has multiplicity of 0..n or 1..n. [JCA80001]
- 1389 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the  
1390 component does not have a reference with the name supplied in the referenceName parameter.  
1391 [JCA80011]
- 1392 • The ComponentContext.getService method MUST throw an IllegalArgumentException if the service  
1393 reference with the name supplied in the referenceName does not have an interface compatible with  
1394 the interface supplied in the businessInterface parameter. [JCA80012]

1395

1396 **getServiceReference ( Class<B> businessInterface, String referenceName ) method:**

1397 Returns a ServiceReference object for a reference defined by the current component, where the  
1398 reference has multiplicity 0..1 or 1..1.

1399 Returns:

- 1400 • **ServiceReference<B>** which is a ServiceReference proxy object for the reference, which implements  
1401 the interface contained in the businessInterface parameter.

1402 The ComponentContext.getServiceReference method MUST return a ServiceReference object typed  
1403 by the interface provided by the businessInterface parameter, for the reference named by the  
1404 referenceName parameter with the interface defined by the businessInterface parameter when that  
1405 reference has a target service configured. [JCA80013]

1406 The ComponentContext.getServiceReference method MUST return null if the multiplicity of the  
1407 reference named by the referenceName parameter is 0..1 and the reference has no target service  
1408 configured. [JCA80007]

1409 Parameters:

- 1410 • **Class<B> businessInterface** - the Java interface for the service reference
- 1411 • **String referenceName** - the name of the service reference

1412 Exceptions:

- 1413 • The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if  
1414 the reference named by the referenceName parameter has multiplicity greater than one. [JCA80004]
- 1415 • The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if  
1416 the reference named by the referenceName parameter does not have an interface of the type defined  
1417 by the businessInterface parameter. [JCA80005]

1418 • The `ComponentContext.getServiceReference` method MUST throw an `IllegalArgumentException` if  
 1419 the component does not have a reference with the name provided in the `referenceName` parameter.  
 1420 [JCA80006]  
 1421

1422 **`getServices(Class<B> businessInterface, String referenceName)` method:**  
 1423 Returns a list of typed service proxies for a reference defined by the current component, where the  
 1424 reference has multiplicity 0..n or 1..n.  
 1425 Returns:

1426 • **`Collection<B>`** which is a collection of proxy objects for the reference, one for each target service to  
 1427 which the reference is wired, where each proxy object implements the interface `B` contained in the  
 1428 `businessInterface` parameter.

1429 The `ComponentContext.getServices` method MUST return a collection containing one proxy object  
 1430 implementing the interface provided by the `businessInterface` parameter for each of the target  
 1431 services configured on the reference identified by the `referenceName` parameter. [JCA80014]

1432 The `ComponentContext.getServices` method MUST return an empty collection if the service reference  
 1433 with the name supplied in the `referenceName` parameter is not wired to any target services.  
 1434 [JCA80015]

1435 Parameters:

1436 • **`Class<B> businessInterface`** - the Java interface for the service reference  
 1437 • **`String referenceName`** - the name of the service reference

1438 Exceptions:

1439 • The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the  
 1440 reference identified by the `referenceName` parameter has multiplicity of 0..1 or 1..1. [JCA80016]  
 1441 • The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the  
 1442 component does not have a reference with the name supplied in the `referenceName` parameter.  
 1443 [JCA80017]  
 1444 • The `ComponentContext.getServices` method MUST throw an `IllegalArgumentException` if the service  
 1445 reference with the name supplied in the `referenceName` parameter does not have an interface compatible with  
 1446 the interface supplied in the `businessInterface` parameter. [JCA80018]  
 1447

1448 **`getServiceReferences(Class<B> businessInterface, String referenceName)` method:**  
 1449 Returns a list of typed `ServiceReference` objects for a reference defined by the current component, where  
 1450 the reference has multiplicity 0..n or 1..n.  
 1451 Returns:

1452 • **`Collection<ServiceReference<B>>`** which is a collection of `ServiceReference` objects for the  
 1453 reference, one for each target service to which the reference is wired, where each proxy object  
 1454 implements the interface `B` contained in the `businessInterface` parameter. The collection is empty if  
 1455 the reference is not wired to any target services.

1456 The `ComponentContext.getServiceReferences` method MUST return a collection containing one  
 1457 `ServiceReference` object typed by the interface provided by the `businessInterface` parameter for each  
 1458 of the target services configured on the reference identified by the `referenceName` parameter.  
 1459 [JCA80019]

1460 The `ComponentContext.getServiceReferences` method MUST return an empty collection if the  
 1461 service reference with the name supplied in the `referenceName` parameter is not wired to any target  
 1462 services. [JCA80020]

1463 Parameters:

1464 • **`Class<B> businessInterface`** - the Java interface for the service reference

1465 • **String referenceName** - the name of the service reference

1466 Exceptions:

1467 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if  
1468 the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1. [JCA80021]

1469 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if  
1470 the component does not have a reference with the name supplied in the referenceName parameter.  
1471 [JCA80022]

1472 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if  
1473 the service reference with the name supplied in the referenceName does not have an interface  
1474 compatible with the interface supplied in the businessInterface parameter. [JCA80023]

1475

1476 **createSelfReference(Class<B> businessInterface) method:**

1477 Returns a ServiceReference object that can be used to invoke this component over the designated  
1478 service.

1479 Returns:

1480 • **ServiceReference<B>** which is a ServiceReference object for the service of this component which  
1481 has the supplied business interface. If the component has multiple services with the same business  
1482 interface the SCA runtime can return a ServiceReference for any one of them.

1483 The ComponentContext.createSelfReference method MUST return a ServiceReference object typed  
1484 by the interface defined by the businessInterface parameter for one of the services of the invoking  
1485 component which has the interface defined by the businessInterface parameter. [JCA80024]

1486 Parameters:

1487 • **Class<B> businessInterface** - the Java interface for the service

1488 Exceptions:

1489 • The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if  
1490 the component does not have a service which implements the interface identified by the  
1491 businessInterface parameter. [JCA80025]

1492

1493 **createSelfReference(Class<B> businessInterface, String serviceName) method:**

1494 Returns a ServiceReference that can be used to invoke this component over the designated service. The  
1495 serviceName parameter explicitly declares the service name to invoke

1496 Returns:

1497 • **ServiceReference<B>** which is a ServiceReference proxy object for the reference, which implements  
1498 the interface contained in the businessInterface parameter.

1499 The ComponentContext.createSelfReference method MUST return a ServiceReference object typed  
1500 by the interface defined by the businessInterface parameter for the service identified by the  
1501 serviceName of the invoking component and which has the interface defined by the businessInterface  
1502 parameter. [JCA80026]

1503 Parameters:

1504 • **Class<B> businessInterface** - the Java interface for the service reference

1505 • **String serviceName** - the name of the service reference

1506 Exceptions:

1507 • The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the  
1508 component does not have a service with the name identified by the serviceName parameter.  
1509 [JCA80027]

- 1510 • The `ComponentContext.createSelfReference` method MUST throw an `IllegalArgumentException` if the  
1511 component service with the name identified by the `serviceName` parameter does not implement a  
1512 business interface which is compatible with the supplied `businessInterface` parameter. [JCA80028]

1513

1514 ***getProperty (Class<B> type, String propertyName) method:***

1515 Returns the value of an SCA property defined by this component.

1516 Returns:

- 1517 • **<B>** which is an object of the type identified by the type parameter containing the value specified for  
1518 the property in the SCA configuration of the component. *null* if the SCA configuration of the  
1519 component does not specify any value for the property.

1520 The `ComponentContext.getProperty` method MUST return an object of the type identified by the type  
1521 parameter containing the value specified in the component configuration for the property named by  
1522 the `propertyName` parameter or null if no value is specified in the configuration. [JCA80029]

1523 Parameters:

- 1524 • **Class<B> type** - the Java class of the property (Object mapped type for primitive Java types - e.g.  
1525 Integer if the type is int)
- 1526 • **String propertyName** - the name of the property

1527 Exceptions:

- 1528 • The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the  
1529 component does not have a property with the name identified by the `propertyName` parameter.  
1530 [JCA80030]
- 1531 • The `ComponentContext.getProperty` method MUST throw an `IllegalArgumentException` if the  
1532 component property with the name identified by the `propertyName` parameter does not have a type  
1533 which is compatible with the supplied type parameter. [JCA80031]

1534

1535 ***getRequestContext() method:***

1536 Returns the `RequestContext` for the current SCA service request.

1537 Returns:

- 1538 • **RequestContext** which is the `RequestContext` object for the current SCA service invocation. *null* if  
1539 there is no current request or if the context is unavailable.

1540 The `ComponentContext.getRequestContext` method MUST return non-null when invoked during the  
1541 execution of a Java business method for a service operation or a callback operation, on the same  
1542 thread that the SCA runtime provided, and MUST return null in all other cases. [JCA80002]

1543 Parameters:

- 1544 • **none**

1545 Exceptions:

- 1546 • **none**

1547

1548 ***cast(B target) method:***

1549 Casts a type-safe reference to a `ServiceReference`

1550 Returns:

- 1551 • **ServiceReference<B>** which is a `ServiceReference` object which implements the same business  
1552 interface B as a reference proxy object

1553 The `ComponentContext.cast` method MUST return a `ServiceReference` object which is typed by the  
1554 same business interface as specified by the reference proxy object supplied in the `target` parameter.  
1555 [JCA80032]

1556 Parameters:

- 1557 • **B target** - a type safe reference proxy object which implements the business interface B

1558 Exceptions:

- 1559 • The `ComponentContext.cast` method MUST throw an `IllegalArgumentException` if the supplied target  
1560 parameter is not an SCA reference proxy object. [JCA80033]

1561 A component can access its component context by defining a field or setter method typed by  
1562 **`org.oasisopen.sca.ComponentContext`** and annotated with **`@Context`**. To access a target service, the  
1563 component uses **`ComponentContext.getService(..)`**.

1564 Snippet 9-1 shows an example of component context usage in a Java class using the `@Context`  
1565 annotation.

```
1566 private ComponentContext componentContext;  
1567  
1568 @Context  
1569 public void setContext(ComponentContext context) {  
1570     componentContext = context;  
1571 }  
1572  
1573 public void doSomething() {  
1574     HelloWorld service =  
1575         componentContext.getService(HelloWorld.class, "HelloWorldComponent");  
1576     service.hello("hello");  
1577 }
```

1578 *Snippet 9-1: ComponentContext Injection Example*

1579 Similarly, non-SCA client code can use the `ComponentContext` API to perform operations against a  
1580 component in an SCA domain. How the non-SCA client code obtains a reference to a `ComponentContext`  
1581 is runtime specific.

## 1582 9.2 Request Context

1583 Figure 9-2 shows the **`RequestContext`** interface:

1584

```
1585 package org.oasisopen.sca;  
1586  
1587 import javax.security.auth.Subject;  
1588  
1589 public interface RequestContext {  
1590  
1591     Subject getSecuritySubject();  
1592  
1593     String getServiceName();  
1594     <CB> ServiceReference<CB> getCallbackReference();  
1595     <CB> CB getCallback();  
1596     <B> ServiceReference<B> getServiceReference();  
1597 }
```

1598 *Figure 9-2: RequestContext Interface*

1599

1600 **`getSecuritySubject ( )` method:**

1601 Returns the JAAS Subject of the current request (see [the JAAS Reference Guide \[JAAS\]](#) for details of  
1602 JAAS).

1603 Returns:

- 1604 • **`javax.security.auth.Subject`** object which is the JAAS subject for the request.  
1605 **`null`** if there is no subject for the request.

1606 The RequestContext.getSecuritySubject method MUST return the JAAS subject of the current  
 1607 request, or null if there is no subject or null if the method is invoked from code not processing a  
 1608 service request or callback request. [JCA80034]

1609 Parameters:

- 1610 • **none**

1611 Exceptions:

- 1612 • **none**

1613

1614 **getServiceName ( ) method:**

1615 Returns the name of the service on the Java implementation the request came in on.

1616 Returns:

- 1617 • **String** containing the name of the service. **null** if the method is invoked from a thread that is not  
 1618 processing a service operation or a callback operation.

1619 The RequestContext.getServiceName method MUST return the name of the service for which an  
 1620 operation is being processed, or null if invoked from a thread that is not processing a service  
 1621 operation or a callback operation. [JCA80035]

1622 Parameters:

- 1623 • **none**

1624 Exceptions:

- 1625 • **none**

1626

1627 **getCallbackReference ( ) method:**

1628 Returns a service reference proxy for the callback for the invoked service operation, as specified by the  
 1629 service client.

1630 Returns:

- 1631 • **ServiceReference<CB>** which is a service reference for the callback for the invoked service, as  
 1632 supplied by the service client. It is typed with the callback interface.

1633 **null** if the invoked service has an interface which is not bidirectional or if the getCallbackReference()  
 1634 method is called during the processing of a callback operation.

1635 **null** if the method is invoked from a thread that is not processing a service operation.

1636 The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by  
 1637 the interface of the callback supplied by the client of the invoked service, or null if either the invoked  
 1638 service is not bidirectional or if the method is invoked from a thread that is not processing a service  
 1639 operation. [JCA80036]

1640 Parameters:

- 1641 • **none**

1642 Exceptions:

- 1643 • **none**

1644

1645 **getCallback ( ) method:**

1646 Returns a proxy for the callback for the invoked service as specified by the service client.

1647 Returns:

- 1648 • **CB** proxy object for the callback for the invoked service as supplied by the service client. It is typed  
 1649 with the callback interface.



**null** if the invoked service has an interface which is not bidirectional or if the `getCallback()` method is called during the processing of a callback operation.

**null** if the method is invoked from a thread that is not processing a service operation.

The `RequestContext.getCallback` method MUST return a reference proxy object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation. [JCA80037]

Parameters:

- **none**

Exceptions:

- **none**

#### **`getServiceReference ( )` method:**

Returns a `ServiceReference` object for the service that was invoked.

Returns:

- **`ServiceReference<B>`** which is a service reference for the invoked service. It is typed with the interface of the service.

**null** if the method is invoked from a thread that is not processing a service operation or a callback operation.

When invoked during the execution of a service operation, the `RequestContext.getServiceReference` method MUST return a `ServiceReference` that represents the service that was invoked. [JCA80003]

When invoked during the execution of a callback operation, the `RequestContext.getServiceReference` method MUST return a `ServiceReference` that represents the callback that was invoked. [JCA80038]

When invoked from a thread not involved in the execution of either a service operation or of a callback operation, the `RequestContext.getServiceReference` method MUST return null. [JCA80039]

Parameters:

- **none**

Exceptions:

- **none**

`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.

## **9.3 ServiceReference Interface**

`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.

Figure 9-3 defines the **`ServiceReference`** interface:

```
package org.oasisopen.sca;

public interface ServiceReference<B> extends java.io.Serializable {

    B getService();
    Class<B> getBusinessInterface();
}
```

Figure 9-3: ServiceReference Interface

**getService ( ) method:**

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.

Returns:

- **<B>** which is type-safe reference proxy object to the target of this reference. It is typed with the interface of the target service.

The ServiceReference.getService method MUST return a reference proxy object which can be used to invoke operations on the target service of the reference and which is typed with the business interface of the reference. [JCA80040]

Parameters:

- **none**

Exceptions:

- **none**

**getBusinessInterface ( ) method:**

Returns the Java class for the business interface associated with this ServiceReference.

Returns:

- **Class<B>** which is a Class object of the business interface associated with the reference.

The ServiceReference.getBusinessInterface method MUST return a Class object representing the business interface of the reference. [JCA80041]

Parameters:

- **none**

Exceptions:

- **none**

## 9.4 ResponseDispatch interface

The **ResponseDispatch** interface is shown in Figure 9-4:

```
package org.oasisopen.sca;

public interface ResponseDispatch<T> {
    void sendResponse(T res);
    void sendFault(Throwable e);
    Map<String, Object> getContext();
}
```

Figure 9-4: ResponseDispatch Interface

**sendResponse ( T response ) method:**

Sends the response message from an asynchronous service method. This method can only be invoked once for a given ResponseDispatch object and cannot be invoked if sendFault has previously been invoked for the same ResponseDispatch object.

Returns:

- 1740 • **void**  
1741 The `ResponseDispatch.sendResponse()` method MUST send the response message to the client of  
1742 an asynchronous service. [JCA50057]

1743 Parameters:

- 1744 • **T** - an instance of the response message returned by the service operation

1745 Exceptions:

- 1746 • The `ResponseDispatch.sendResponse()` method MUST throw an `InvalidStateException` if either the  
1747 `sendResponse` method or the `sendFault` method has already been called once. [JCA80058]

1748

1749 ***sendFault ( Throwable e ) method:***

1750 Sends an exception as a fault from an asynchronous service method. This method can only be invoked  
1751 once for a given `ResponseDispatch` object and cannot be invoked if `sendResponse` has previously been  
1752 invoked for the same `ResponseDispatch` object.

1753 Returns:

- 1754 • **void**

1755 The `ResponseDispatch.sendFault()` method MUST send the supplied fault to the client of an  
1756 asynchronous service. [JCA80059]

1757 Parameters:

- 1758 • **e** - an instance of an exception returned by the service operation

1759 Exceptions:

- 1760 • The `ResponseDispatch.sendFault()` method MUST throw an `InvalidStateException` if either the  
1761 `sendResponse` method or the `sendFault` method has already been called once. [JCA80060]

1762

1763 ***getContext () method:***

1764 Obtains the context object for the `ResponseDispatch` method

1765 Returns:

- 1766 • **Map<String, object>** which is the context object for the `ResponseDispatch` object.  
1767 The invoker can update the context object with appropriate context information, prior to invoking  
1768 either the `sendResponse` method or the `sendFault` method

1769 Parameters:

- 1770 • **none**

1771 Exceptions:

- 1772 • **none**

## 1773 9.5 ServiceRuntimeException

1774 Figure 9-5 shows the ***ServiceRuntimeException***.

1775

```
1776 package org.oasisopen.sca;  
1777  
1778 public class ServiceRuntimeException extends RuntimeException {  
1779     ...  
1780 }
```

1781 Figure 9-5: *ServiceRuntimeException*

1782

1783 This exception signals problems in the management of SCA component execution.

## 9.6 ServiceUnavailableException

Figure 9-6 shows the **ServiceUnavailableException**.

```
package org.oasisopen.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    ...
}
```

Figure 9-6: ServiceUnavailableException

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

## 9.7 InvalidServiceException

Figure 9-7 shows the **InvalidServiceException**.

```
package org.oasisopen.sca;

public class InvalidServiceException extends ServiceRuntimeException {
    ...
}
```

Figure 9-7: InvalidServiceException

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.

## 9.8 Constants

The SCA **Constants** interface defines a number of constant values that are used in the SCA Java APIs and Annotations. Figure 9-8 shows the Constants interface:

```
package org.oasisopen.sca;

public interface Constants {

    String SCA_NS = "http://docs.oasis-open.org/ns/opencsa/sca/200912";

    String SCA_PREFIX = "{"+SCA_NS+"}";

    String SERVERAUTHENTICATION = SCA_PREFIX + "serverAuthentication";
    String CLIENTAUTHENTICATION = SCA_PREFIX + "clientAuthentication";
    String ATLEASTONCE = SCA_PREFIX + "atLeastOnce";
    String ATMOSTONCE = SCA_PREFIX + "atMostOnce";
    String EXACTLYONCE = SCA_PREFIX + "exactlyOnce";
    String ORDERED = SCA_PREFIX + "ordered";
    String TRANSACTEDONEWAY = SCA_PREFIX + "transactedOneWay";
    String IMMEDIATEONEWAY = SCA_PREFIX + "immediateOneWay";
    String PROPAGATESTransaction = SCA_PREFIX + "propagatesTransaction";
    String SUSPENDSTransaction = SCA_PREFIX + "suspendsTransaction";
    String ASYNCINVOcation = SCA_PREFIX + "asyncInvocation";
    String SOAP = SCA_PREFIX + "SOAP";
}
```

```

String JMS = SCA_PREFIX + "JMS";
String NOLISTENER = SCA_PREFIX + "noListener";
String EJB = SCA_PREFIX + "EJB";
}

```

Figure 9-8: Constants Interface

## 9.9 SCAClientFactory Class

The SCAClientFactory class provides the means for client code to obtain a proxy reference object for a service within an SCA Domain, through which the client code can invoke operations of that service. This is particularly useful for client code that is running outside the SCA Domain containing the target service, for example where the code is "unmanaged" and is not running under an SCA runtime.

The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods which the client can invoke in order to obtain a concrete object implementing the SCAClientFactory interface for a particular SCA Domain. The returned SCAClientFactory object provides a getService() method which provides the client with the means to obtain a reference proxy object for a service running in the SCA Domain.

The SCAClientFactory class is shown in Figure 9-9:

```

package org.oasisopen.sca.client;

import java.net.URI;
import java.util.Properties;

import org.oasisopen.sca.NoSuchDomainException;
import org.oasisopen.sca.NoSuchServiceException;
import org.oasisopen.sca.client.SCAClientFactoryFinder;
import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

public abstract class SCAClientFactory {

    protected static SCAClientFactoryFinder factoryFinder;

    private URI domainURI;

    private SCAClientFactory() {
    }

    protected SCAClientFactory(URI domainURI)
        throws NoSuchDomainException {
        this.domainURI = domainURI;
    }

    protected URI getDomainURI() {
        return domainURI;
    }

    public static SCAClientFactory newInstance( URI domainURI )
        throws NoSuchDomainException {
        return newInstance(null, null, domainURI);
    }

    public static SCAClientFactory newInstance(Properties properties,
                                                URI domainURI)
        throws NoSuchDomainException {
        return newInstance(properties, null, domainURI);
    }

    public static SCAClientFactory newInstance(ClassLoader classLoader,

```

```

1892                                     URI domainURI)
1893     throws NoSuchDomainException {
1894     return newInstance(null, classLoader, domainURI);
1895 }
1896
1897 public static SCAClientFactory newInstance(Properties properties,
1898                                     ClassLoader classLoader,
1899                                     URI domainURI)
1900     throws NoSuchDomainException {
1901     final SCAClientFactoryFinder finder =
1902         factoryFinder != null ? factoryFinder :
1903             new SCAClientFactoryFinderImpl();
1904     final SCAClientFactory factory
1905         = finder.find(properties, classLoader, domainURI);
1906     return factory;
1907 }
1908
1909 public abstract <T> T getService(Class<T> interfaze, String serviceURI)
1910     throws NoSuchServiceException, NoSuchDomainException;
1911 }

```

Figure 9-9: SCAClientFactory Class

#### ***newInstance ( URI domainURI ) method:***

Obtains a object implementing the SCAClientFactory class.

Returns:

- **object** which implements the SCAClientFactory class

The SCAClientFactory.newInstance( URI ) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80042]

Parameters:

- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance( URI ) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80043]

#### ***newInstance(Properties properties, URI domainURI) method:***

Obtains a object implementing the SCAClientFactory class, using a specified set of properties.

Returns:

- **object** which implements the SCAClientFactory class

The SCAClientFactory.newInstance( Properties, URI ) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter. [JCA80044]

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory class.
- **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

Exceptions:

- The SCAClientFactory.newInstance( Properties, URI ) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain. [JCA80045]

1942 ***newInstance(Classloader classLoader, URI domainURI) method:***

1943 Obtains a object implementing the SCAClientFactory class using a specified classloader.

1944 Returns:

1945 • **object** which implements the SCAClientFactory class

1946 The SCAClientFactory.newInstance( Classloader, URI ) method MUST return an object which

1947 implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.

1948 [JCA80046]

1949 Parameters:

1950 • **classLoader** - a ClassLoader to use when creating the object which implements the

1951 SCAClientFactory class.

1952 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

1953 Exceptions:

1954 • The SCAClientFactory.newInstance( Classloader, URI ) method MUST throw a

1955 NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.

1956 [JCA80047]

1957

1958 ***newInstance(Properties properties, Classloader classLoader, URI domainURI) method:***

1959 Obtains a object implementing the SCAClientFactory class using a specified set of properties and a

1960 specified classloader.

1961 Returns:

1962 • **object** which implements the SCAClientFactory class

1963 The SCAClientFactory.newInstance( Properties, Classloader, URI ) method MUST return an object

1964 which implements the SCAClientFactory class for the SCA Domain identified by the domainURI

1965 parameter. [JCA80048]

1966 Parameters:

1967 • **properties** - a set of Properties that can be used when creating the object which implements the

1968 SCAClientFactory class.

1969 • **classLoader** - a ClassLoader to use when creating the object which implements the

1970 SCAClientFactory class.

1971 • **domainURI** - a URI for the SCA Domain which is targeted by the returned SCAClient object

1972 Exceptions:

1973 • The SCAClientFactory.newInstance( Properties, Classloader, URI ) MUST throw a

1974 NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.

1975 [JCA80049]

1976

1977 ***getService( Class<T> interfaze, String serviceURI ) method:***

1978 Obtains a proxy reference object for a specified target service in a specified SCA Domain.

1979 Returns:

1980 • **<T>** a proxy object which implements the business interface T

1981 Invocations of a business method of the proxy causes the invocation of the corresponding operation

1982 of the target service.

1983 The SCAClientFactory.getService method MUST return a proxy object which implements the

1984 business interface defined by the interfaze parameter and which can be used to invoke operations on

1985 the service identified by the serviceURI parameter. [JCA80050]

1986 Parameters:

1987 • **interfaze** - a Java interface class which is the business interface of the target service

1988 • **serviceURI** - a String containing the relative URI of the target service within its SCA Domain.  
 1989 Takes the form componentName/serviceName or can also take the extended form  
 1990 componentName/serviceName/bindingName to use a specific binding of the target service  
 1991 Exceptions:  
 1992 • The SCAClientFactory.getService method MUST throw a NoSuchServiceException if a service with  
 1993 the relative URI serviceURI and a business interface which matches interface cannot be found in the  
 1994 SCA Domain targeted by the SCAClient object. [JCA80051]  
 1995 • The SCAClientFactory.getService method MUST throw a NoSuchServiceException if the domainURI  
 1996 of the SCAClientFactory does not identify a valid SCA Domain. [JCA80052]  
 1997  
 1998 **SCAClientFactory ( URI ) method:** a single argument constructor that must be available on all concrete  
 1999 subclasses of SCAClientFactory. The URI required is the URI of the Domain targeted by the  
 2000 SCAClientFactory  
 2001  
 2002 **getDomainURI() method:**  
 2003 Obtains the Domain URI value for this SCAClientFactory  
 2004 Returns:  
 2005 • **URI** of the target SCA Domain for this SCAClientFactory  
 2006 The SCAClientFactory.getDomainURI method MUST return the SCA Domain URI of the Domain  
 2007 associated with the SCAClientFactory object. [JCA80053]  
 2008 Parameters:  
 2009 • **none**  
 2010 Exceptions:  
 2011 • The SCAClientFactory.getDomainURI method MUST throw a **NoSuchServiceException** if the  
 2012 domainURI of the SCAClientFactory does not identify a valid SCA Domain. [JCA80054]  
 2013  
 2014 **private SCAClientFactory() method:**  
 2015 This private no-argument constructor prevents instantiation of an SCAClientFactory instance without the  
 2016 use of the constructor with an argument, even by subclasses of the abstract SCAClientFactory class.  
 2017  
 2018 **factoryFinder protected field:**  
 2019 Provides a means by which a provider of an SCAClientFactory implementation can inject a factory finder  
 2020 implementation into the abstract SCAClientFactory class - once this is done, future invocations of the  
 2021 SCAClientFactory use the injected factory finder to locate and return an instance of a subclass of  
 2022 SCAClientFactory.

## 2023 9.10 SCAClientFactoryFinder Interface

2024 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory  
 2025 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can  
 2026 create alternative implementations of this interface that use different class loading or lookup mechanisms:  
 2027

```
2028 package org.oasisopen.sca.client;
2029
2030 public interface SCAClientFactoryFinder {
2031
2032     SCAClientFactory find(Properties properties,
2033                          ClassLoader classLoader,
2034                          URI domainURI )
```



```
throws NoSuchDomainException ;
}
```

Figure 9-10: SCAClientFactoryFinder Interface

**find (Properties properties, ClassLoader classloader, URI domainURI) method:**

Obtains an implementation of the SCAClientFactory interface.

Returns:

- **SCAClientFactory** implementation object

The implementation of the SCAClientFactoryFinder.find method MUST return an object which is an implementation of the SCAClientFactory interface, for the SCA Domain represented by the domainURI parameter, using the supplied properties and classloader. [JCA80055]

Parameters:

- **properties** - a set of Properties that can be used when creating the object which implements the SCAClientFactory interface.
- **classLoader** - a ClassLoader to use when creating the object which implements the SCAClientFactory interface.
- **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

Exceptions:

- The implementation of the SCAClientFactoryFinder.find method MUST throw a ServiceRuntimeException if the SCAClientFactory implementation could not be found. [JCA80056]

## 9.11 SCAClientFactoryFinderImpl Class

This class is a default implementation of an SCAClientFactoryFinder, which is used to find an implementation of an SCAClientFactory subclass, as used to obtain an SCAClient object for use by a client. SCA runtime providers can replace this implementation with their own version.

```
package org.oasisopen.sca.client.impl;

public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
    ...
    public SCAClientFactoryFinderImpl() {...}

    public SCAClientFactory find(Properties properties,
                                ClassLoader classLoader
                                URI domainURI)
    throws NoSuchDomainException, ServiceRuntimeException {...}
    ...
}
```

Snippet 9-2: SCAClientFactoryFinderImpl Class

**SCAClientFactoryFinderImpl () method:**

Public constructor for the SCAClientFactoryFinderImpl.

Returns:

- **SCAClientFactoryFinderImpl** which implements the SCAClientFactoryFinder interface

Parameters:

- **none**

Exceptions:

2081 • **none**

2082

2083 **find (Properties, ClassLoader, URI) method:**

2084 Obtains an implementation of the SCAClientFactory interface. It discovers a provider's SCAClientFactory

2085 implementation by referring to the following information in this order:

2086 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the

2087 newInstance() method call if specified

2088 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties

2089 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

2090 Returns:

2091 • **SCAClientFactory** implementation object

2092 Parameters:

2093 • **properties** - a set of Properties that can be used when creating the object which implements the

2094 SCAClientFactory interface.

2095 • **classLoader** - a ClassLoader to use when creating the object which implements the

2096 SCAClientFactory interface.

2097 • **domainURI** - a URI for the SCA Domain targeted by the SCAClientFactory

2098 Exceptions:

2099 • **ServiceRuntimeException** - if the SCAClientFactory implementation could not be found

## 2100 9.12 NoSuchDomainException

2101 Figure 9-11 shows the **NoSuchDomainException**:

2102

```
2103 package org.oasisopen.sca;
2104
2105 public class NoSuchDomainException extends Exception {
2106     ...
2107 }
```

2108 *Figure 9-11: NoSuchDomainException Class*

2109

2110 This exception indicates that the Domain specified could not be found.

## 2111 9.13 NoSuchServiceException

2112 Figure 9-12 shows the **NoSuchServiceException**:

2113

```
2114 package org.oasisopen.sca;
2115
2116 public class NoSuchServiceException extends Exception {
2117     ...
2118 }
```

2119 *Figure 9-12: NoSuchServiceException Class*

2120

2121 This exception indicates that the service specified could not be found.

## 10 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 the sections "`@Property`" and "`@Reference`" 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]

### 10.1 @AllowsPassByReference

Figure 10-1 defines the `@AllowsPassByReference` 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 java.lang.annotation.Retention;
import java.lang.annotation.Target;

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

    boolean value() default true;
}
```

Figure 10-1: `AllowsPassByReference` Annotation

The `@AllowsPassByReference` annotation allows service method implementations and client references to be marked as "allows pass by reference" to indicate that they use input parameters, return values and exceptions in a manner that allows the SCA runtime to avoid the cost of copying mutable objects when a remotable service is called locally within the same JVM.

The `@AllowsPassByReference` annotation has the attribute:

- **value** – specifies whether the "allows pass by reference" marker applies to the service implementation class, service implementation method, or client reference to which this annotation applies; if not specified, defaults to true.

The `@AllowsPassByReference` annotation MUST only annotate the following locations:

- a service implementation class
- an individual method of a remotable service implementation
- an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter [JCA90052]

The "allows pass by reference" marking of a method implementation of a remotable service is determined as follows:

- 2169 1. If the method has an `@AllowsPassByReference` annotation, the method is marked “allows pass by  
2170 reference” if and only if the value of the method’s annotation is true.
- 2171 2. Otherwise, if the class has an `@AllowsPassByReference` annotation, the method is marked “allows  
2172 pass by reference” if and only if the value of the class’s annotation is true.
- 2173 3. Otherwise, the method is not marked “allows pass by reference”.
- 2174 The “allows pass by reference” marking of a reference for a remotable service is determined as follows:
- 2175 1. If the reference has an `@AllowsPassByReference` annotation, the reference is marked “allows pass  
2176 by reference” if and only if the value of the reference’s annotation is true.
- 2177 2. Otherwise, if the service implementation class containing the reference has an  
2178 `@AllowsPassByReference` annotation, the reference is marked “allows pass by reference” if and only  
2179 if the value of the class’s annotation is true.
- 2180 3. Otherwise, the reference is not marked “allows pass by reference”.
- 2181 Snippet 10-1 shows a sample where `@AllowsPassByReference` is defined for the implementation of a  
2182 service method on the Java component implementation class.

2183

```
2184 @AllowsPassByReference  
2185 public String hello(String message) {  
2186     ...  
2187 }
```

2188 *Snippet 10-1: Use of `@AllowsPassByReference` on a Method*

2189

2190 Snippet 10-2 shows a sample where `@AllowsPassByReference` is defined for a client reference of a Java  
2191 component implementation class.

2192

```
2193 @AllowsPassByReference  
2194 @Reference  
2195 private StockQuoteService stockQuote;
```

2196 *Snippet 10-2: Use of `@AllowsPassByReference` on a Reference*

## 2197 10.2 `@AsyncFault`

2198 Figure 10-2 defines the `@AsyncFault` annotation:

2199

```
2200 package org.oasisopen.sca.annotation;  
2201  
2202 import static java.lang.annotation.ElementType.METHOD;  
2203 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2204  
2205 import java.lang.annotation.Inherited;  
2206 import java.lang.annotation.Retention;  
2207 import java.lang.annotation.Target;  
2208  
2209 @Inherited  
2210 @Target({METHOD})  
2211 @Retention(RUNTIME)  
2212 public @interface AsyncFault {  
2213     Class<?>[] value() default {};  
2214  
2215  
2216 }
```

2217 *Figure 10-2: `AsyncFault` Annotation*

2218

2219 The **@AsyncFault** annotation is used to indicate the faults/exceptions which are returned by the  
2220 asynchronous service method which it annotates.

## 2221 10.3 @AsyncInvocation

2222 Figure 10-3 defines the **@AsyncInvocation** annotation, which is used to attach the "asyncInvocation"  
2223 policy intent to an interface or to a method:  
2224

```
2225 package org.oasisopen.sca.annotation;  
2226  
2227 import static java.lang.annotation.ElementType.METHOD;  
2228 import static java.lang.annotation.ElementType.TYPE;  
2229 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2230 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2231  
2232 import java.lang.annotation.Inherited;  
2233 import java.lang.annotation.Retention;  
2234 import java.lang.annotation.Target;  
2235  
2236 @Inherited  
2237 @Target({TYPE, METHOD})  
2238 @Retention(RUNTIME)  
2239 @Intent(AsyncInvocation.ASYNCINVOCATION)  
2240 public @interface AsyncInvocation {  
2241     String ASYNCINVOCATION = SCA_PREFIX + "asyncInvocation";  
2242  
2243     boolean value() default true;  
2244 }
```

2245 *Figure 10-3: AsyncInvocation Annotation*

2246  
2247 The **@AsyncInvocation** annotation is used to indicate that the operations of a Java interface uses the  
2248 long-running request-response pattern as described in the SCA Assembly specification.

## 2249 10.4 @Authentication

2250 The following Java code defines the **@Authentication** annotation:  
2251

```
2252 package org.oasisopen.sca.annotation;  
2253  
2254 import static java.lang.annotation.ElementType.FIELD;  
2255 import static java.lang.annotation.ElementType.METHOD;  
2256 import static java.lang.annotation.ElementType.PARAMETER;  
2257 import static java.lang.annotation.ElementType.TYPE;  
2258 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2259 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2260  
2261 import java.lang.annotation.Inherited;  
2262 import java.lang.annotation.Retention;  
2263 import java.lang.annotation.Target;  
2264  
2265 @Inherited  
2266 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2267 @Retention(RUNTIME)  
2268 @Intent(Authentication.AUTHENTICATION)  
2269 public @interface Authentication {  
2270     String AUTHENTICATION = SCA_PREFIX + "authentication";  
2271     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";  
2272     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";  
2273 }
```

```

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

```

Figure 10-4: Authentication Annotation

The **@Authentication** annotation is used to indicate the need for authentication. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

## 10.5 @Authorization

Figure 10-5 defines the @Authorization 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;

/**
 * The @Authorization annotation is used to indicate that
 * an authorization policy is required.
 */
@Inherited
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
@Intent(Authorization.AUTHORIZATION)
public @interface Authorization {
    String AUTHORIZATION = SCA_PREFIX + "authorization";
}

```

Figure 10-5: Authorization Annotation

The **@Authorization** annotation is used to indicate the need for an authorization policy. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

## 10.6 @Callback

Figure 10-6 defines the **@Callback** annotation:

```

package org.oasisopen.sca.annotation;

```

```

2325 import static java.lang.annotation.ElementType.FIELD;
2326 import static java.lang.annotation.ElementType.METHOD;
2327 import static java.lang.annotation.ElementType.TYPE;
2328 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2329 import java.lang.annotation.Retention;
2330 import java.lang.annotation.Target;
2331
2332 @Target({TYPE, METHOD, FIELD})
2333 @Retention(RUNTIME)
2334 public @interface Callback {
2335
2336     Class<?> value() default Void.class;
2337 }

```

Figure 10-6: Callback Annotation

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

The @Callback annotation has the 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] When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class. [JCA90054] When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface. [JCA90058]

The @Callback annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope. [JCA90057]

Snippet 10-3 shows an example use of the @Callback annotation to declare a callback interface.

```

2359 package somepackage;
2360 import org.oasisopen.sca.annotation.Callback;
2361 import org.oasisopen.sca.annotation.Remotable;
2362 @Remotable
2363 @Callback(MyServiceCallback.class)
2364 public interface MyService {
2365
2366     void someMethod(String arg);
2367 }
2368
2369 @Remotable
2370 public interface MyServiceCallback {
2371
2372     void receiveResult(String result);
2373 }

```

Snippet 10-3: Use of @Callback

The implied component type is for Snippet 10-3 is shown in Snippet 10-4.

```

<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" >
  <service name="MyService">
    <interface.java interface="somepackage.MyService"
      callbackInterface="somepackage.MyServiceCallback"/>
  </service>
</componentType>

```

Snippet 10-4: Implied componentType for Snippet 10-3

## 10.7 @ComponentName

Figure 10-7 defines the **@ComponentName** 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.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

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

```

Figure 10-7: ComponentName Annotation

The @ComponentName annotation is used to denote a Java class field or setter method that is used to inject the component name.

Snippet 10-5 shows a component name field definition sample.

```

@ComponentName
private String componentName;

```

Snippet 10-5: Use of @ComponentName on a Field

Snippet 10-6 shows a component name setter method sample.

```

@ComponentName
public void setComponentName(String name) {
  //...
}

```

Snippet 10-6: Use of @ComponentName on a Setter

## 10.8 @Confidentiality

Figure 10-8 defines the **@Confidentiality** 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;

```



```

2427 import static java.lang.annotation.ElementType.TYPE;
2428 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2429 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2430
2431 import java.lang.annotation.Inherited;
2432 import java.lang.annotation.Retention;
2433 import java.lang.annotation.Target;
2434
2435 @Inherited
2436 @Target({TYPE, FIELD, METHOD, PARAMETER})
2437 @Retention(RUNTIME)
2438 @Intent(Confidentiality.CONFIDENTIALITY)
2439 public @interface Confidentiality {
2440     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
2441     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
2442     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
2443
2444     /**
2445      * List of confidentiality qualifiers such as "message" or
2446      * "transport".
2447      *
2448      * @return confidentiality qualifiers
2449      */
2450     @Qualifier
2451     String[] value() default "";
2452 }

```

Figure 10-8: Confidentiality Annotation

The **@Confidentiality** annotation is used to indicate the need for confidentiality. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

## 10.9 @Constructor

Figure 10-9 defines the **@Constructor** annotation:

```

2461 package org.oasisopen.sca.annotation;
2462
2463 import static java.lang.annotation.ElementType.CONSTRUCTOR;
2464 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2465 import java.lang.annotation.Retention;
2466 import java.lang.annotation.Target;
2467
2468 @Target (CONSTRUCTOR)
2469 @Retention(RUNTIME)
2470 public @interface Constructor { }

```

Figure 10-9: Constructor Annotation

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]

Snippet 10-7 shows a sample for the **@Constructor** annotation.

```

2479 public class HelloServiceImpl implements HelloService {
2480
2481     public HelloServiceImpl() {
2482         ...
2483     }
2484
2485     @Constructor
2486     public HelloServiceImpl(@Property(name="someProperty")
2487                             String someProperty ) {
2488         ...
2489     }
2490
2491     public String hello(String message) {
2492         ...
2493     }
2494 }

```

Snippet 10-7: Use of @Constructor

## 10.10 @Context

Figure 10-10 defines the **@Context** annotation:

```

2499 package org.oasisopen.sca.annotation;
2500
2501 import static java.lang.annotation.ElementType.FIELD;
2502 import static java.lang.annotation.ElementType.METHOD;
2503 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2504 import java.lang.annotation.Retention;
2505 import java.lang.annotation.Target;
2506
2507 @Target({METHOD, FIELD})
2508 @Retention(RUNTIME)
2509 public @interface Context {
2510
2511 }

```

Figure 10-10: Context 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.

Snippet 10-8 shows a ComponentContext field definition sample.

```

2521 @Context
2522 protected ComponentContext context;

```

Snippet 10-8: Use of @Context for a ComponentContext

Snippet 10-9 shows a RequestContext field definition sample.

```

2527 @Context
2528 protected RequestContext context;

```

Snippet 10-9: Use of @Context for a RequestContext

## 10.11 @Destroy

Figure 10-11 defines the **@Destroy** annotation:

```
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 Destroy {
}
```

Figure 10-11: Destroy Annotation

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 can have any access modifier and MUST have a void return type and no arguments. [JCA90004]

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. [JCA90005]

Snippet 10-10 shows a sample for a destroy method definition.

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

Snippet 10-10: Use of @Destroy

## 10.12 @EagerInit

Figure 10-12: EagerInit Annotation defines the **@EagerInit** 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 EagerInit {
}
```

Figure 10-12: EagerInit Annotation

2577 The **@EagerInit** annotation is used to mark the Java class of a COMPOSITE scoped implementation for  
2578 eager initialization. When marked for eager initialization with an @EagerInit annotation, the composite  
2579 scoped instance MUST be created when its containing component is started. [JCA90007]

## 2580 10.13 @Init

2581 Figure 10-13: Init Annotation defines the **@Init** annotation:  
2582

```
2583 package org.oasisopen.sca.annotation;  
2584  
2585 import static java.lang.annotation.ElementType.METHOD;  
2586 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2587 import java.lang.annotation.Retention;  
2588 import java.lang.annotation.Target;  
2589  
2590 @Target(METHOD)  
2591 @Retention(RUNTIME)  
2592 public @interface Init {  
2593  
2594  
2595 }
```

2596 Figure 10-13: Init Annotation

2597  
2598 The @Init annotation is used to denote a single Java class method that is called when the scope defined  
2599 for the implementation class starts. A method marked with the @Init annotation can have any access  
2600 modifier and MUST have a void return type and no arguments. [JCA90008]  
2601 If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime  
2602 MUST call the annotated method after all property and reference injection is complete. [JCA90009]  
2603 Snippet 10-11 shows an example of an init method definition.  
2604

```
2605 @Init  
2606 public void myInitMethod() {  
2607     ...  
2608 }
```

2609 Snippet 10-11: Use of @Init

## 2610 10.14 @Integrity

2611 Figure 10-14 defines the **@Integrity** annotation:  
2612

```
2613 package org.oasisopen.sca.annotation;  
2614  
2615 import static java.lang.annotation.ElementType.FIELD;  
2616 import static java.lang.annotation.ElementType.METHOD;  
2617 import static java.lang.annotation.ElementType.PARAMETER;  
2618 import static java.lang.annotation.ElementType.TYPE;  
2619 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2620 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2621  
2622 import java.lang.annotation.Inherited;  
2623 import java.lang.annotation.Retention;  
2624 import java.lang.annotation.Target;  
2625  
2626 @Inherited  
2627 @Target({TYPE, FIELD, METHOD, PARAMETER})
```

```

2628 @Retention(RUNTIME)
2629 @Intent(Integrity.INTEGRITY)
2630 public @interface Integrity {
2631     String INTEGRITY = SCA_PREFIX + "integrity";
2632     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
2633     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
2634
2635     /**
2636      * List of integrity qualifiers (such as "message" or "transport").
2637      *
2638      * @return integrity qualifiers
2639      */
2640     @Qualifier
2641     String[] value() default "";
2642 }

```

Figure 10-14: Integrity Annotation

The **@Integrity** annotation is used to indicate that the invocation requires integrity (i.e. no tampering of the messages between client and service). See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

## 10.15 @Intent

Figure 10-15 defines the **@Intent** annotation:

```

2652 package org.oasisopen.sca.annotation;
2653
2654 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
2655 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2656 import java.lang.annotation.Retention;
2657 import java.lang.annotation.Target;
2658
2659 @Target({ANNOTATION_TYPE})
2660 @Retention(RUNTIME)
2661 public @interface Intent {
2662     /**
2663      * The qualified name of the intent, in the form defined by
2664      * {@link javax.xml.namespace.QName#toString}.
2665      * @return the qualified name of the intent
2666      */
2667     String value() default "";
2668
2669     /**
2670      * The XML namespace for the intent.
2671      * @return the XML namespace for the intent
2672      */
2673     String targetNamespace() default "";
2674
2675     /**
2676      * The name of the intent within its namespace.
2677      * @return name of the intent within its namespace
2678      */
2679     String localPart() default "";
2680 }

```

Figure 10-15: Intent Annotation

2683 The @Intent annotation is used for the creation of new annotations for specific intents. It is not expected  
2684 that the @Intent annotation will be used in application code.

2685 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to define new  
2686 intent annotations.

## 2687 10.16 @ManagedSharedTransaction

2688 Figure 10-16 defines the @ManagedSharedTransaction annotation:

2689

```
2690 package org.oasisopen.sca.annotation;  
2691  
2692 import static java.lang.annotation.ElementType.FIELD;  
2693 import static java.lang.annotation.ElementType.METHOD;  
2694 import static java.lang.annotation.ElementType.PARAMETER;  
2695 import static java.lang.annotation.ElementType.TYPE;  
2696 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2697 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2698  
2699 import java.lang.annotation.Inherited;  
2700 import java.lang.annotation.Retention;  
2701 import java.lang.annotation.Target;  
2702  
2703 /**  
2704  * The @ManagedSharedTransaction annotation is used to indicate that  
2705  * a distributed ACID transaction is required.  
2706  */  
2707 @Inherited  
2708 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2709 @Retention(RUNTIME)  
2710 @Intent(ManagedSharedTransaction.MANAGEDSHAREDTRANSACTION)  
2711 public @interface ManagedSharedTransaction {  
2712     String MANAGEDSHAREDTRANSACTION = SCA_PREFIX + "managedSharedTransaction";  
2713 }
```

2714 Figure 10-16: ManagedSharedTransaction Annotation

2715

2716 The **@ManagedSharedTransaction** annotation is used to indicate the need for a distributed and globally  
2717 coordinated ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the  
2718 meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent  
2719 annotations are used in Java.

## 2720 10.17 @ManagedTransaction

2721 Figure 10-17 defines the @ManagedTransaction annotation:

2722

```
2723 import static java.lang.annotation.ElementType.FIELD;  
2724 import static java.lang.annotation.ElementType.METHOD;  
2725 import static java.lang.annotation.ElementType.PARAMETER;  
2726 import static java.lang.annotation.ElementType.TYPE;  
2727 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2728 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2729  
2730 import java.lang.annotation.Inherited;  
2731 import java.lang.annotation.Retention;  
2732 import java.lang.annotation.Target;  
2733  
2734 /**  
2735  * The @ManagedTransaction annotation is used to indicate the
```

```

2736 * need for an ACID transaction environment.
2737 */
2738 @Inherited
2739 @Target({TYPE, FIELD, METHOD, PARAMETER})
2740 @Retention(RUNTIME)
2741 @Intent(ManagedTransaction.MANAGEDTRANSACTION)
2742 public @interface ManagedTransaction {
2743     String MANAGEDTRANSACTION = SCA_PREFIX + "managedTransaction";
2744     String MANAGEDTRANSACTION_MESSAGELOCAL = MANAGEDTRANSACTION + ".local";
2745     String MANAGEDTRANSACTION_TRANSPORTGLOBAL = MANAGEDTRANSACTION +
2746     ".global";
2747
2748     /**
2749      * List of managedTransaction qualifiers (such as "global" or "local").
2750      *
2751      * @return managedTransaction qualifiers
2752      */
2753     @Qualifier
2754     String[] value() default "";
2755 }

```

Figure 10-17: ManagedTransaction Annotation

The **@ManagedTransaction** annotation is used to indicate the need for an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations are used in Java.

## 10.18 @MutualAuthentication

Figure 10-18 defines the @MutualAuthentication annotation:

```

2764 package org.oasisopen.sca.annotation;
2765
2766 import static java.lang.annotation.ElementType.FIELD;
2767 import static java.lang.annotation.ElementType.METHOD;
2768 import static java.lang.annotation.ElementType.PARAMETER;
2769 import static java.lang.annotation.ElementType.TYPE;
2770 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2771 import static org.oasisopen.sca.Constants.SCA_PREFIX;
2772
2773 import java.lang.annotation.Inherited;
2774 import java.lang.annotation.Retention;
2775 import java.lang.annotation.Target;
2776
2777 /**
2778  * The @MutualAuthentication annotation is used to indicate that
2779  * a mutual authentication policy is needed.
2780  */
2781 @Inherited
2782 @Target({TYPE, FIELD, METHOD, PARAMETER})
2783 @Retention(RUNTIME)
2784 @Intent(MutualAuthentication.MUTUALAUTHENTICATION)
2785 public @interface MutualAuthentication {
2786     String MUTUALAUTHENTICATION = SCA_PREFIX + "mutualAuthentication";
2787 }

```

Figure 10-18: MutualAuthentication Annotation

The **@MutualAuthentication** annotation is used to indicate the need for mutual authentication between a service consumer and a service provider. See the SCA Policy Framework Specification [POLICY] for

2792 details on the meaning of the intent. See the [section on Application of Intent Annotations](#) for samples of  
2793 how intent annotations are used in Java.

## 2794 10.19 @NoManagedTransaction

2795 Figure 10-19 defines the @NoManagedTransaction annotation:  
2796

```
2797 package org.oasisopen.sca.annotation;  
2798  
2799 import static java.lang.annotation.ElementType.FIELD;  
2800 import static java.lang.annotation.ElementType.METHOD;  
2801 import static java.lang.annotation.ElementType.PARAMETER;  
2802 import static java.lang.annotation.ElementType.TYPE;  
2803 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2804 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
2805  
2806 import java.lang.annotation.Inherited;  
2807 import java.lang.annotation.Retention;  
2808 import java.lang.annotation.Target;  
2809  
2810 /**  
2811  * The @NoManagedTransaction annotation is used to indicate that  
2812  * a non-transactional environment is needed.  
2813  */  
2814 @Inherited  
2815 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2816 @Retention(RUNTIME)  
2817 @Intent(NoManagedTransaction.NOMANAGEDTRANSACTION)  
2818 public @interface NoManagedTransaction {  
2819     String NOMANAGEDTRANSACTION = SCA_PREFIX + "noManagedTransaction";  
2820 }
```

2821 *Figure 10-19: NoManagedTransaction Annotation*

2822  
2823 The **@NoManagedTransaction** annotation is used to indicate that the component does not want to run in  
2824 an ACID transaction. See the SCA Policy Framework Specification [POLICY] for details on the meaning  
2825 of the intent. See the [section on Application of Intent Annotations](#) for samples of how intent annotations  
2826 are used in Java.

## 2827 10.20 @OneWay

2828 Figure 10-20 defines the **@OneWay** annotation:  
2829

```
2830 package org.oasisopen.sca.annotation;  
2831  
2832 import static java.lang.annotation.ElementType.METHOD;  
2833 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2834 import java.lang.annotation.Retention;  
2835 import java.lang.annotation.Target;  
2836  
2837 @Target(METHOD)  
2838 @Retention(RUNTIME)  
2839 public @interface OneWay {  
2840  
2841  
2842 }
```

2843 *Figure 10-20: OneWay Annotation*



2844

2845 A method annotated with `@OneWay` MUST have a void return type and MUST NOT have declared  
2846 checked exceptions. [JCA90055]

2847 When a method of a Java interface is annotated with `@OneWay`, the SCA runtime MUST ensure that all  
2848 invocations of that method are executed in a non-blocking fashion, as described in the section on  
2849 Asynchronous Programming. [JCA90056]

2850 The `@OneWay` annotation has no attributes.

2851 Snippet 10-12 shows the use of the `@OneWay` annotation on an interface.

2852

```
2853 package services.hello;  
2854  
2855 import org.oasisopen.sca.annotation.OneWay;  
2856  
2857 public interface HelloService {  
2858     @OneWay  
2859     void hello(String name);  
2860 }
```

2861 *Snippet 10-12: Use of `@OneWay`*

## 2862 10.21 `@PolicySets`

2863 Figure 10-21 defines the `@PolicySets` annotation:

2864

```
2865 package org.oasisopen.sca.annotation;  
2866  
2867 import static java.lang.annotation.ElementType.FIELD;  
2868 import static java.lang.annotation.ElementType.METHOD;  
2869 import static java.lang.annotation.ElementType.PARAMETER;  
2870 import static java.lang.annotation.ElementType.TYPE;  
2871 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2872  
2873 import java.lang.annotation.Retention;  
2874 import java.lang.annotation.Target;  
2875  
2876 @Target({TYPE, FIELD, METHOD, PARAMETER})  
2877 @Retention(RUNTIME)  
2878 public @interface PolicySets {  
2879     /**  
2880      * Returns the policy sets to be applied.  
2881      *  
2882      * @return the policy sets to be applied  
2883      */  
2884     String[] value() default "";  
2885 }
```

2886 *Figure 10-21: PolicySets Annotation*

2887

2888 The `@PolicySets` annotation is used to attach one or more SCA Policy Sets to a Java implementation  
2889 class or to one of its subelements.

2890 See the [section "Policy Set Annotations"](#) for details and samples.

## 2891 10.22 `@Property`

2892 Figure 10-22 defines the `@Property` annotation:

2893

```

2894 package org.oasisopen.sca.annotation;
2895
2896 import static java.lang.annotation.ElementType.FIELD;
2897 import static java.lang.annotation.ElementType.METHOD;
2898 import static java.lang.annotation.ElementType.PARAMETER;
2899 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2900 import java.lang.annotation.Retention;
2901 import java.lang.annotation.Target;
2902
2903 @Target({METHOD, FIELD, PARAMETER})
2904 @Retention(RUNTIME)
2905 public @interface Property {
2906
2907     String name() default "";
2908     boolean required() default true;
2909 }

```

Figure 10-22: Property Annotation

2910  
2911

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.

2916 When the Java type of a field, setter method or constructor parameter with the @Property annotation is a  
2917 primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by  
2918 an SCA component definition into an instance of the Java type as defined by the XML to Java mapping in  
2919 the JAXB specification [JAXB] with XML schema validation enabled. [JCA90061]

2920 When the Java type of a field, setter method or constructor parameter with the @Property annotation is  
2921 not a JAXB annotated class, the SCA runtime can use any XML to Java mapping when converting  
2922 property values into instances of the Java type.

2923 The @Property annotation MUST NOT be used on a class field that is declared as final. [JCA90011]

2924 Where there is both a setter method and a field for a property, the setter method is used.

2925 The @Property annotation has the attributes:

2926 • **name (0..1)** – the name of the property. For a field annotation, the default is the name of the field of  
2927 the Java class. For a setter method annotation, the default is the JavaBeans property name  
2928 [JAVABEANS] corresponding to the setter method name. For a @Property annotation applied to a  
2929 constructor parameter, there is no default value for the name attribute and the name attribute MUST  
2930 be present. [JCA90013]

2931 • **required (0..1)** – a boolean value which specifies whether injection of the property value is required  
2932 or not, where true means injection is required and false means injection is not required. Defaults to  
2933 true. For a @Property annotation applied to a constructor parameter, the required attribute MUST  
2934 NOT have the value false. [JCA90014]

2935

2936 Snippet 10-13 shows a property field definition sample.

2937

```

2938 @Property(name="currency", required=true)
2939 protected String currency;
2940
2941 The following snippet shows a property setter sample
2942
2943 @Property(name="currency", required=true)
2944 public void setCurrency( String theCurrency ) {
2945     ....
2946 }

```

Snippet 10-13: Use of @Property on a Field

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 SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false. [JCA90047]

Snippet 10-14 shows the definition of a configuration property using the @Property annotation for a collection.

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

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

Snippet 10-14: Use of @Property with a Collection

## 10.23 @Qualifier

Figure 10-23 defines the @Qualifier annotation:

```
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 Qualifier {
}
```

Figure 10-23: Qualifier Annotation

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.

## 10.24 @Reference

Figure 10-24 defines the @Reference 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.RetentionPolicy.RUNTIME;
```

```

2997 import java.lang.annotation.Retention;
2998 import java.lang.annotation.Target;
2999 @Target({METHOD, FIELD, PARAMETER})
3000 @Retention(RUNTIME)
3001 public @interface Reference {
3002
3003     String name() default "";
3004     boolean required() default true;
3005 }

```

Figure 10-24: Reference Annotation

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]

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

The @Reference annotation has the attributes:

- **name : String (0..1)** – 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 (0..1)** – 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]

Snippet 10-15 shows a reference field definition sample.

```

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

```

Snippet 10-15: Use of @Reference on a Field

Snippet 10-16 shows a reference setter sample

```

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

```

Snippet 10-16: Use of @Reference on a Setter

Snippet 10-17 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.

```

package services.hello;

private HelloService helloService;

@Reference(name="helloService", required=true)

```

```

3047 public setHelloService(HelloService service) {
3048     helloService = service;
3049 }
3050
3051 public void clientMethod() {
3052     String result = helloService.hello("Hello World!");
3053     ...
3054 }

```

3055 *Snippet 10-17: Use of @Reference and a ServiceReference*

3056  
3057 The presence of a @Reference annotation is reflected in the componentType information that the runtime  
3058 generates through reflection on the implementation class. Snippet 10-18 shows the component type for  
3059 the component implementation fragment in Snippet 10-17.

```

3061 <?xml version="1.0" encoding="ASCII"?>
3062 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">
3063
3064     <!-- Any services offered by the component would be listed here -->
3065     <reference name="helloService" multiplicity="1..1">
3066         <interface.java interface="services.hello.HelloService"/>
3067     </reference>
3068
3069 </componentType>

```

3070 *Snippet 10-18: Implied componentType for Implementation in Snippet 10-17*

3071  
3072 If the type of a reference is not an array or any type that extends or implements java.util.Collection, then  
3073 the SCA runtime MUST introspect the component type of the implementation with a <reference/> element  
3074 with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with  
3075 @multiplicity=1..1 if the @Reference annotation required attribute is true. [JCA90020]

3076 If the type of a reference is defined as an array or as any type that extends or implements  
3077 java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation  
3078 with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is  
3079 false and with @multiplicity=1..n if the @Reference annotation required attribute is true. [JCA90021]

3080 Snippet 10-19 shows a sample of a service reference definition using the @Reference annotation on a  
3081 java.util.List. The name of the reference is "helloServices" and its type is HelloService. The clientMethod()  
3082 calls the "hello" operation of all the services referenced by the helloServices reference. In this case, at  
3083 least one HelloService needs to be present, so **required** is true.

```

3085 @Reference(name="helloServices", required=true)
3086 protected List<HelloService> helloServices;
3087
3088 public void clientMethod() {
3089
3090     ...
3091     for (int index = 0; index < helloServices.size(); index++) {
3092         HelloService helloService =
3093             (HelloService)helloServices.get(index);
3094         String result = helloService.hello("Hello World!");
3095     }
3096     ...
3097 }

```

3098 *Snippet 10-19: Use of @Reference with a List of ServiceReferences*

Snippet 10-20 shows the XML representation of the component type reflected from 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 version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912">

  <!-- Any services offered by the component would be listed here -->
  <reference name="helloServices" multiplicity="1..n">
    <interface.java interface="services.hello.HelloService"/>
  </reference>
</componentType>
```

Snippet 10-20: Implied componentType for Implementation in Snippet 10-19

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]

## 10.24.1 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:

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 MUST either continue to work or throw an InvalidServiceException when it is invoked. [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 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 MUST either continue to work or throw an InvalidServiceException when it is invoked. [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. [JCA90034] If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or

3151 has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,  
3152 and attempts to call business methods SHOULD throw an InvalidServiceException or a  
3153 ServiceUnavailableException. [JCA90035] If the target service of a reference or ServiceReference  
3154 accessed through the component context by calling getService() or getServiceReference() has changed,  
3155 the returned value SHOULD be a reference to the changed service. [JCA90036]

3156 The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This means  
3157 that in the cases where reference reinjection is not allowed, the array or Collection for a reference of  
3158 multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference  
3159 wiring or to the targets of the wiring. [JCA90037] In cases where the contents of a reference array or  
3160 collection change when the wiring changes or the targets change, then for references that use setter  
3161 injection, the setter method MUST be called by the SCA runtime for any change to the contents.  
3162 [JCA90038] A reinjected array or Collection for a reference MUST NOT be the same array or Collection  
3163 object previously injected to the component. [JCA90039]

3164

<u>Change event</u>	<u>Effect on</u>		
	Injected Reference or ServiceReference	Existing ServiceReference Object**	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	can be reinjected (if other conditions* apply). If not reinjected, then it continues to work as if the reference target was not changed.	continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service undeployed	Business methods throw InvalidServiceException.	Business methods throw InvalidServiceException.	Result is a reference to the undeployed service. Business methods throw InvalidServiceException.
Target service becomes unavailable	Business methods throw ServiceUnavailableException	Business methods throw ServiceUnavailableException	Result is be a reference to the unavailable service. Business methods throw ServiceUnavailableException.
Target service changed	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.	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.	Result is a reference to the changed service.
* Other conditions: The component cannot be STATELESS scoped. The reference has to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed. ** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().			

3165 Table 10-1Reinjection Effects

## 10.25 @Remotable

Figure 10-25 defines the `@Remotable` 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, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Remotable {
}
```

Figure 10-25: Remotable Annotation

The `@Remotable` annotation is used to indicate that an SCA service interface is remotable. The `@Remotable` annotation is valid only on a Java interface, a Java class, a field, a setter method, or a constructor parameter. It MUST NOT appear anywhere else. [JCA90053] A remotable service can be published externally as a service and MUST be translatable into a WSDL portType. [JCA90040]

The `@Remotable` annotation has no attributes. When placed on a Java service interface, it indicates that the interface is remotable. When placed on a Java service implementation class, it indicates that all SCA service interfaces provided by the class (including the class itself, if the class defines an SCA service interface) are remotable. When placed on a service reference, it indicates that the interface for the reference is remotable.

Snippet 10-21 shows the Java interface for a remotable service with its `@Remotable` annotation.

```
package services.hello;

import org.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

Snippet 10-21: Use of `@Remotable` on an Interface

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.



3218 Snippet 10-22 shows how a Java service implementation class can use the @Remotable annotation to  
3219 define a remotable SCA service interface using a Java service interface that is not marked as remotable.

3220

```
3221 package services.hello;
3222
3223 import org.oasisopen.sca.annotation.*;
3224
3225 public interface HelloService {
3226
3227     String hello(String message);
3228 }
3229
3230 package services.hello;
3231
3232 import org.oasisopen.sca.annotation.*;
3233
3234 @Remotable
3235 @Service(HelloService.class)
3236 public class HelloServiceImpl implements HelloService {
3237
3238     public String hello(String message) {
3239         ...
3240     }
3241 }
```

3242 *Snippet 10-22: Use of @Remotable on a Class*

3243

3244 Snippet 10-23 shows how a reference can use the @Remotable annotation to define a remotable SCA  
3245 service interface using a Java service interface that is not marked as remotable.

3246

```
3247 package services.hello;
3248
3249 import org.oasisopen.sca.annotation.*;
3250
3251 public interface HelloService {
3252
3253     String hello(String message);
3254 }
3255
3256 package services.hello;
3257
3258 import org.oasisopen.sca.annotation.*;
3259
3260 public class HelloClient {
3261
3262     @Remotable
3263     @Reference
3264     protected HelloService myHello;
3265
3266     public String greeting(String message) {
3267         return myHello.hello(message);
3268     }
3269 }
```

3270 *Snippet 10-23: Use of @Remotable on a Reference*

## 3271 10.26 @Requires

3272 Figure 10-26 defines the @Requires annotation:

3273

```

3274 package org.oasisopen.sca.annotation;
3275
3276 import static java.lang.annotation.ElementType.FIELD;
3277 import static java.lang.annotation.ElementType.METHOD;
3278 import static java.lang.annotation.ElementType.PARAMETER;
3279 import static java.lang.annotation.ElementType.TYPE;
3280 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3281
3282 import java.lang.annotation.Inherited;
3283 import java.lang.annotation.Retention;
3284 import java.lang.annotation.Target;
3285
3286 @Inherited
3287 @Retention(RUNTIME)
3288 @Target({TYPE, METHOD, FIELD, PARAMETER})
3289 public @interface Requires {
3290     /**
3291      * Returns the attached intents.
3292      *
3293      * @return the attached intents
3294      */
3295     String[] value() default "";
3296 }

```

Figure 10-26: Requires Annotation

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.

## 10.27 @Scope

Figure 10-27 defines the **@Scope** annotation:

```

3305 package org.oasisopen.sca.annotation;
3306
3307 import static java.lang.annotation.ElementType.TYPE;
3308 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3309 import java.lang.annotation.Retention;
3310 import java.lang.annotation.Target;
3311
3312 @Target(TYPE)
3313 @Retention(RUNTIME)
3314 public @interface Scope {
3315     String value() default "STATELESS";
3316 }
3317

```

Figure 10-27: Scope Annotation

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 attribute:

- **value** – the name of the scope.
- SCA defines the following scope names, but others can be defined by particular Java-based implementation types

3326 STATELESS

3327 COMPOSITE

3328 The default value is STATELESS.

3329 Snippet 10-24 shows a sample for a COMPOSITE scoped service implementation:

3330

```
3331 package services.hello;
3332
3333 import org.oasisopen.sca.annotation.*;
3334
3335 @Service(HelloService.class)
3336 @Scope("COMPOSITE")
3337 public class HelloServiceImpl implements HelloService {
3338
3339     public String hello(String message) {
3340         ...
3341     }
3342 }
```

3343 *Snippet 10-24: Use of @Scope*

## 3344 10.28 @Service

3345 Figure 10-28 defines the **@Service** annotation:

3346

```
3347 package org.oasisopen.sca.annotation;
3348
3349 import static java.lang.annotation.ElementType.TYPE;
3350 import static java.lang.annotation.RetentionPolicy.RUNTIME;
3351 import java.lang.annotation.Retention;
3352 import java.lang.annotation.Target;
3353
3354 @Target(TYPE)
3355 @Retention(RUNTIME)
3356 public @interface Service {
3357
3358     Class<?>[] value();
3359     String[] names() default {};
3360 }
```

3361 *Figure 10-28: Service Annotation*

3362

3363 The @Service annotation is used on a component implementation class to specify the SCA services  
3364 offered by the implementation. An implementation class need not be declared as implementing all of the  
3365 interfaces implied by the services declared in its @Service annotation, but all methods of all the declared  
3366 service interfaces MUST be present. [JCA90042] A class used as the implementation of a service is not  
3367 required to have a @Service annotation. If a class has no @Service annotation, then the rules  
3368 determining which services are offered and what interfaces those services have are determined by the  
3369 specific implementation type.

3370 The @Service annotation has the attributes:

- 3371
- 3372 • **value (1..1)** – An array of interface or class objects that are exposed as services by this  
implementation. If the array is empty, no services are exposed.
  - 3373 • **names (0..1)** - An array of Strings which are used as the service names for each of the interfaces  
3374 declared in the **value** array. The number of Strings in the names attribute array of the @Service  
3375 annotation MUST match the number of elements in the value attribute array. [JCA90050] The value of

3376 each element in the @Service names array MUST be unique amongst all the other element values in  
3377 the array. [JCA90060]

3378 The **service name** of an exposed service defaults to the name of its interface or class, without the  
3379 package name. If the names attribute is specified, the service name for each interface or class in the  
3380 value attribute array is the String declared in the corresponding position in the names attribute array.

3381 If a component implementation has two services with the same Java simple name, the names attribute of  
3382 the @Service annotation MUST be specified. [JCA90045] If a Java implementation needs to realize two  
3383 services with the same Java simple name then this can be achieved through subclassing of the interface.

3384 Snippet 10-25 shows an implementation of the HelloService marked with the @Service annotation.

3385

```
3386 package services.hello;  
3387  
3388 import org.oasisopen.sca.annotation.Service;  
3389  
3390 @Service(HelloService.class)  
3391 public class HelloServiceImpl implements HelloService {  
3392  
3393     public void hello(String name) {  
3394         System.out.println("Hello " + name);  
3395     }  
3396 }
```

3397 *Snippet 10-25: Use of @Service*

3398

# 11 WSDL to Java and Java to WSDL

3399  
3400  
3401

This specification applies the WSDL to Java and Java to WSDL mapping rules as defined by [the JAX-WS 2.1 specification \[JAX-WS\]](#) for generating remotable Java interfaces from WSDL portTypes and vice versa.

3402  
3403  
3404  
3405  
3406  
3407  
3408

SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL. [\[JCA100022\]](#) 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`. [\[JCA100003\]](#)

3409  
3410  
3411  
3412  
3413  
3414

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 MUST support the JAXB 2.1 mapping from XML Schema to Java and from Java to XML Schema. [\[JCA100004\]](#) SCA runtimes MAY support the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema. [\[JCA100005\]](#) 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.

3415

## 11.1 JAX-WS Annotations and SCA Interfaces

3416  
3417  
3418  
3419  
3420  
3421  
3422  
3423

A Java class or interface used to define an SCA interface can contain JAX-WS annotations. In addition to affecting the Java to WSDL mapping defined by [the JAX-WS specification \[JAX-WS\]](#) these annotations can impact the SCA interface. An SCA runtime MUST apply the JAX-WS annotations as described in [Table 11-1 and Table 11-2](#) when introspecting a Java class or interface class. [\[JCA100011\]](#) This could mean that the interface of a Java implementation is defined by a WSDL interface declaration. [If the services provided by an implementation class are explicitly identified by an @Service annotation, only the identified classes define services of the implementation even if implemented interfaces that are not listed in the @Service annotation contain @JAX-WS annotations.](#)

Annotation	Property	Impact to SCA Interface
@WebService		A Java interface or class annotated with <code>@WebService</code> MUST be treated as if annotated with the SCA <code>@Remotable</code> annotation <a href="#">[JCA100012]</a>
	name	<del>If used to define a service, sets service name</del> <a href="#">The value of the name attribute of the @WebService annotation, if present, MUST be used to define the name of an SCA service when there is no @Service annotation present in the SCA component implementation. [JCA100023]</a>  <a href="#">The value of the name attribute of the @WebService</a>

annotation, if present, MUST be used to define the name of an SCA service when the @Service annotation is present without the names attribute and indicates that the Java interface or class annotated with the @WebService annotation defines an SCA service interface. [JCA100028]

targetNamespace      None

serviceName          None

**wsdlLocation**      A Java class annotated with the @WebService annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class. [JCA100013]

**endpointInterface**      A Java class annotated with the @WebService annotation with its endpointInterface attribute set MUST have its interface defined by the referenced interface instead of annotated Java class. [JCA100014]

portName              None

#### @WebMethod

**operationName**      ~~Sets operation name~~ For a Java method annotated with the @WebMethod annotation with the operationName set, an SCA runtime MUST use the value of the operationName attribute as the SCA operation name. [JCA100024]

action                None

	<b>exclude</b>	Method is excluded from the interface. An SCA runtime MUST NOT include a Java method annotated with the @WebMethod annotation with the exclude attribute set to true in an SCA interface. [JCA100025]
	<b>@OneWay</b>	The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation. [JCA100002]
<b>@WebParam</b>	<b>name</b>	Sets parameter name
	<b>targetNamespace</b>	None
	<b>mode</b>	Sets directionality of parameter. For a Java parameter annotated with the @WebParam annotation with the mode attribute set, an SCA runtime MUST apply the value of the mode attribute when comparing interfaces. [JCA100026]
	<b>header</b>	A Java class or interface containing an @WebParam annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100015]
<b>@WebResult</b>	<b>partName</b>	Overrides name
	<b>name</b>	Sets parameter name
	<b>targetNamespace</b>	None

	header	A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100016]
	partName	Overrides name
	@SOAPBinding	A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface. [JCA100021]
	style	
	use	
	parameterStyle	
	@HandlerChain	None
	file	
	name	

3424 Table 11-1: JSR 181 Annotations and SCA Interfaces

3425

Annotation	Property	Impact to SCA Interface
@ServiceMode		A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class. [JCA100017]
	value	
@WebFault		
	name	<del>Sets fault name</del> None



<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	targetNamespace	None
	faultBean	None
		None
@RequestWrapper	localName	
	targetNamespace	
	className	
@ResponseWrapper		None
	localName	
	targetNamespace	
	className	
@WebServiceClient		An interface or class annotated with
		@WebServiceClient MUST NOT be used to define an
		SCA interface. [JCA100018]
	name	
	targetNamespace	
	wsdlLocation	
@WebEndpoint		None
	name	
@WebServiceProvider		A class annotated with @WebServiceProvider MUST be
		treated as if annotated with the SCA @Remotable
		annotation. [JCA100019]

<i>Annotation</i>	<i>Property</i>	<i>Impact to SCA Interface</i>
	<b>wsdlLocation</b>	A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class. [JCA100020]
	serviceName	None
	portName	None
	targetNamespace	None
@BindingType		None
	value	
@WebServiceRef		See JEE specification
	name	
	wsdlLocation	
	type	
	value	
	mappedName	
@WebServiceRefs		See JEE specification
	value	
@Action		None
	fault	
	input	

Annotation	Property	Impact to SCA Interface
	output	
@FaultAction		None
	value	
	output	

3426 Table 11-2: JSR 224 Annotations and SCA Interfaces

3427 **11.2 JAX-WS Client Asynchronous API for a Synchronous Service**

3428 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client  
 3429 application with a means of invoking that service asynchronously, so that the client can invoke a service  
 3430 operation and proceed to do other work without waiting for the service operation to complete its  
 3431 processing. The client application can retrieve the results of the service either through a polling  
 3432 mechanism or via a callback method which is invoked when the operation completes.

3433 For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the  
 3434 additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006] For  
 3435 SCA reference interfaces defined using interface.java, the SCA runtime MUST support a Java interface  
 3436 which contains the additional client-side asynchronous polling and callback methods defined by JAX-WS.  
 3437 [JCA100007] If the additional client-side asynchronous polling and callback methods defined by JAX-WS  
 3438 are present in the interface which declares the type of a reference in the implementation, SCA Runtimes  
 3439 MUST NOT include these methods in the SCA reference interface in the component type of the  
 3440 implementation. [JCA100008]

3441 The additional client-side asynchronous polling and callback methods defined by JAX-WS are recognized  
 3442 in a Java interface according to the steps:

3443 For each method M in the interface, if another method P in the interface has

- 3444 a. a method name that is M's method name with the characters "Async" appended, and
- 3445 b. the same parameter signature as M, and
- 3446 c. a return type of Response<R> where R is the return type of M

3447 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

3448 For each method M in the interface, if another method C in the interface has

- 3449 a. a method name that is M's method name with the characters "Async" appended, and
- 3450 b. a parameter signature that is M's parameter signature with an additional final parameter of  
 3451 type AsyncHandler<R> where R is the return type of M, and
- 3452 c. a return type of Future<?>

3453 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

3454 As an example, an interface can be defined in WSDL as shown in Snippet 11-1:

3455

```

3456 <!-- WSDL extract -->
3457 <message name="getPrice">
3458   <part name="ticker" type="xsd:string"/>
3459 </message>
3460
3461 <message name="getPriceResponse">
3462   <part name="price" type="xsd:float"/>
  
```

```

3463 </message>
3464
3465 <portType name="StockQuote">
3466   <operation name="getPrice">
3467     <input message="tns:getPrice"/>
3468     <output message="tns:getPriceResponse"/>
3469   </operation>
3470 </portType>

```

3471 *Snippet 11-1: Example WSDL Interface*

3472  
3473 The JAX-WS asynchronous mapping will produce the Java interface in Snippet 11-2:

```

3475 // asynchronous mapping
3476 @WebService
3477 public interface StockQuote {
3478   float getPrice(String ticker);
3479   Response<Float> getPriceAsync(String ticker);
3480   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
3481 }

```

3482 *Snippet 11-2: JAX-WS Asynchronous Interface for WSDL Interface in Snippet 11-1*

3483  
3484 For SCA interface definition purposes, this is treated as equivalent to the interface in Snippet 11-3:

```

3486 // synchronous mapping
3487 @WebService
3488 public interface StockQuote {
3489   float getPrice(String ticker);
3490 }

```

3491 *Snippet 11-3: Equivalent SCA Interface Corresponding to Java Interface in Snippet 11-2*

3492  
3493 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] If the  
3494 client implementation uses the asynchronous form of the interface, the two additional getPriceAsync()  
3495 methods can be used for polling and callbacks as defined by the JAX-WS specification.

## 3496 11.3 Treatment of SCA Asynchronous Service API

3497 **For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface**  
3498 **which contains the server-side asynchronous methods defined by SCA.** [JCA100010]

3499 Asynchronous service methods are identified as described in the section "Asynchronous handling of Long  
3500 Running Service Operations" and are mapped to WSDL in the same way as the equivalent synchronous  
3501 method described in that section.

3502 Generating an asynchronous service method from a WSDL request/response operation follows the  
3503 algorithm described in the same section.

---

## 12 Conformance

The XML schema pointed to by the RDDDL document at the namespace URI, defined by this specification, are considered to be authoritative and take precedence over the XML schema defined in the appendix of this document.

Normative code artifacts related to this specification are considered to be authoritative and take precedence over specification text.

There are three categories of artifacts for which this specification defines conformance:

- a) SCA Java XML Document,
- b) SCA Java Class
- c) SCA Runtime.

### 12.1 SCA Java XML Document

An SCA Java XML document is an SCA Composite Document, or an SCA ComponentType Document, as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), that uses the <interface.java> element. Such an SCA Java XML document MUST be a conformant SCA Composite Document or SCA ComponentType Document, as defined by the [SCA Assembly Model specification \[ASSEMBLY\]](#), and MUST comply with the requirements specified in the [Interface section](#) of this specification.

### 12.2 SCA Java Class

An SCA Java Class is a Java class or interface that complies with Java Standard Edition version 5.0 and MAY include annotations and APIs defined in this specification. An SCA Java Class that uses annotations and APIs defined in this specification MUST comply with the requirements specified in this specification for those annotations and APIs.

### 12.3 SCA Runtime

The APIs and annotations defined in this specification are meant to be used by Java-based component implementation models in either partial or complete fashion. A Java-based component implementation specification that uses this specification specifies which of the APIs and annotations defined here are used. The APIs and annotations an SCA Runtime has to support depends on which Java-based component implementation specification the runtime supports. For example, see the [SCA POJO Component Implementation Specification \[JAVA\\_CII\]](#).

An implementation that claims to conform to this specification MUST meet the following conditions:

1. The implementation MUST meet all the conformance requirements defined by the SCA Assembly Model Specification [ASSEMBLY].
2. The implementation MUST support <interface.java> and MUST comply with all the normative statements in Section 3.
3. The implementation MUST reject an SCA Java XML Document that does not conform to the sca-interface-java.xsd schema.
4. The implementation MUST support and comply with all the normative statements in Section 10.

## A. XML Schema: sca-interface-java-1.1.xsd

```

3541 <?xml version="1.0" encoding="UTF-8"?>
3542 <!-- Copyright(C) OASIS(R) 2005,2010. All Rights Reserved.
3543 OASIS trademark, IPR and other policies apply. -->
3544 <schema xmlns="http://www.w3.org/2001/XMLSchema"
3545 targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3546 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
3547 elementFormDefault="qualified">
3548
3549 <include schemaLocation="sca-core-1.1-ed05cd06.xsd"/>
3550
3551 <!-- Java Interface -->
3552 <element name="interface.java" type="sca:JavaInterface"
3553 substitutionGroup="sca:interface"/>
3554 <complexType name="JavaInterface">
3555 <complexContent>
3556 <extension base="sca:Interface">
3557 <sequence>
3558 <any namespace="##other" processContents="lax" minOccurs="0"
3559 maxOccurs="unbounded"/>
3560 </sequence>
3561 <attribute name="interface" type="NCName" use="required"/>
3562 <attribute name="callbackInterface" type="NCName"
3563 use="optional"/>
3564 </extension>
3565 </complexContent>
3566 </complexType>
3567
3568 </schema>

```

---

## B. Java Classes and Interfaces

### B.1 SCAClient Classes and Interfaces

#### B.1.1 SCAClientFactory Class

SCA provides an abstract base class SCAClientFactory. Vendors can provide subclasses of this class which create objects that implement the SCAClientFactory class suitable for linking to services in their SCA runtime.

```
/*
 * Copyright(C) OASIS(R) 2005,2009-2010. All Rights Reserved.
 * OASIS trademark, IPR and other policies apply.
 */
package org.oasisopen.sca.client;

import java.net.URI;
import java.util.Properties;

import org.oasisopen.sca.NoSuchDomainException;
import org.oasisopen.sca.NoSuchServiceException;
import org.oasisopen.sca.client.SCAClientFactoryFinder;
import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

/**
 * The SCAClientFactory can be used by non-SCA managed code to
 * lookup services that exist in a SCADomain.
 *
 * @see SCAClientFactoryFinderImpl
 * * @see SCAClient
 *
 * @author OASIS Open
 */
public abstract class SCAClientFactory {

    /**
     * The SCAClientFactoryFinder.
     * Provides a means by which a provider of an SCAClientFactory
     * implementation can inject a factory finder implementation into
     * the abstract SCAClientFactory class - once this is done, future
     * invocations of the SCAClientFactory use the injected factory
     * finder to locate and return an instance of a subclass of
     * SCAClientFactory.
     */
    protected static SCAClientFactoryFinder factoryFinder;

    /**
     * The Domain URI of the SCA Domain which is accessed by this
     * SCAClientFactory
     */
    private URI domainURI;

    /**
     * Prevent concrete subclasses from using the no-arg constructor
     */
    private SCAClientFactory() {
    }

    /**
```

```

3625     * Constructor used by concrete subclasses
3626     * @param domainURI - The Domain URI of the Domain accessed via this
3627     * SCAClientFactory
3628     */
3629     protected SCAClientFactory(URI domainURI) +
3630     throws NoSuchDomainException {
3631         this.domainURI = domainURI;
3632     }
3633
3634     /**
3635     * Gets the Domain URI of the Domain accessed via this SCAClientFactory
3636     * @return - the URI for the Domain
3637     */
3638     protected URI getDomainURI() {
3639         return domainURI;
3640     }
3641
3642
3643     /**
3644     * Creates a new instance of the SCAClientSCAClientFactory that can be
3645     * used to lookup SCA Services.
3646     *
3647     * @param domainURI      URI of the target domain for the
3648     SCAClientSCAClientFactory
3649     * @return A new SCAClientSCAClientFactory
3650     */
3651     public static SCAClientFactory newInstance( URI domainURI )
3652     throws NoSuchDomainException {
3653         return newInstance(null, null, domainURI);
3654     }
3655
3656     /**
3657     * Creates a new instance of the SCAClientSCAClientFactory that can be
3658     * used to lookup SCA Services.
3659     *
3660     * @param properties      Properties that may be used when
3661     * creating a new instance of the SCAClientSCAClientFactory
3662     * @param domainURI      URI of the target domain for the
3663     SCAClientSCAClientFactory
3664     * @return A new SCAClientSCAClientFactory instance
3665     */
3666     public static SCAClientFactory newInstance(Properties properties,
3667                                                URI domainURI)
3668     throws NoSuchDomainException {
3669         return newInstance(properties, null, domainURI);
3670     }
3671
3672     /**
3673     * Creates a new instance of the SCAClientSCAClientFactory that can be
3674     * used to lookup SCA Services.
3675     *
3676     * @param classLoader      ClassLoader that may be used when
3677     * creating a new instance of the SCAClientSCAClientFactory
3678     * @param domainURI      URI of the target domain for the
3679     SCAClientSCAClientFactory
3680     * @return A new SCAClientSCAClientFactory instance
3681     */
3682     public static SCAClientFactory newInstance(ClassLoader classLoader,
3683                                                URI domainURI)
3684     throws NoSuchDomainException {
3685         return newInstance(null, classLoader, domainURI);
3686     }
3687
3688     /**

```



```

3689 | * Creates a new instance of the SCAClientSCAClientFactory that can be
3690 | * used to lookup SCA Services.
3691 | *
3692 | * @param properties Properties that may be used when
3693 | * creating a new instance of the SCAClientSCAClientFactory
3694 | * @param classLoader ClassLoader that may be used when
3695 | * creating a new instance of the SCAClientSCAClientFactory
3696 | * @param domainURI URI of the target domain for the
3697 | SCAClientSCAClientFactory
3698 | * @return A new SCAClientSCAClientFactory instance
3699 | */
3700 | public static SCAClientFactory newInstance(Properties properties,
3701 |                                           ClassLoader classLoader,
3702 |                                           URI domainURI)
3703 |     throws NoSuchDomainException {
3704 |     final SCAClientFactoryFinder finder =
3705 |         factoryFinder != null ? factoryFinder :
3706 |             new SCAClientFactoryFinderImpl();
3707 |     final SCAClientFactory factory
3708 |         = finder.find(properties, classLoader, domainURI);
3709 |     return factory;
3710 | }
3711 |
3712 | /**
3713 | * Returns a reference proxy that implements the business interface <T>
3714 | * of a service in the SCA Domain handled by this SCAClientFactory
3715 | *
3716 | * @param serviceURI the relative URI of the target service. Takes the
3717 | * form componentName/serviceName.
3718 | * Can also take the extended form componentName/serviceName/bindingName
3719 | * to use a specific binding of the target service
3720 | *
3721 | * @param interfaze The business interface class of the service in the
3722 | * domain
3723 | * @param <T> The business interface class of the service in the domain
3724 | *
3725 | * @return a proxy to the target service, in the specified SCA Domain
3726 | * that implements the business interface <B>.
3727 | * @throws NoSuchServiceException Service requested was not found
3728 | * @throws NoSuchDomainException Domain requested was not found
3729 | */
3730 | public abstract <T> T getService(Class<T> interfaze, String serviceURI)
3731 |     throws NoSuchServiceException, NoSuchDomainException;
3732 | }

```

## 3733 B.1.2 SCAClientFactoryFinder interface

3734 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory  
3735 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can  
3736 create alternative implementations of this interface that use different class loading or lookup mechanisms.

```

3737 |
3738 | /*
3739 | * Copyright (C) OASIS(R) 2005, 2009-2010. All Rights Reserved.
3740 | * OASIS trademark, IPR and other policies apply.
3741 | */
3742 |
3743 | package org.oasisopen.sca.client;
3744 |
3745 | import java.net.URI;
3746 | import java.util.Properties;
3747 |
3748 | import org.oasisopen.sca.NoSuchDomainException;
3749 |

```

```

3750  /* A Service Provider Interface representing a SCAClientFactory finder.
3751  * SCA provides a default reference implementation of this interface.
3752  * SCA runtime vendors can create alternative implementations of this
3753  * interface that use different class loading or lookup mechanisms.
3754  */
3755  public interface SCAClientFactoryFinder {
3756
3757      /**
3758       * Method for finding the SCAClientFactory for a given Domain URI using
3759       * a specified set of properties and a specified ClassLoader
3760       * @param properties - properties to use - may be null
3761       * @param classLoader - ClassLoader to use - may be null
3762       * @param domainURI - the Domain URI - must be a valid SCA Domain URI
3763       * @return - the SCAClientFactory or null if the factory could not be
3764       * @throws - NoSuchDomainException if the domainURI does not reference
3765       * a valid SCA Domain
3766       * found
3767       */
3768       SCAClientFactory find(Properties properties,
3769                           ClassLoader classLoader,
3770                           URI domainURI )
3771       throws NoSuchDomainException ;
3772  }

```

### 3773 B.1.3 SCAClientFactoryFinderImpl class

3774 This class provides a default implementation for finding a provider's SCAClientFactory implementation  
3775 class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the  
3776 base SCAClientFactory class.

3777 It discovers a provider's SCAClientFactory implementation by referring to the following information in this  
3778 order:

- 3779 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the  
3780 newInstance() method call if specified
- 3781 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 3782 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```

3783
3784  /*
3785  * Copyright (C) OASIS (R) 2005, 2009-2010. All Rights Reserved.
3786  * OASIS trademark, IPR and other policies apply.
3787  */
3788  package org.oasisopen.sca.client.impl;
3789
3790  import org.oasisopen.sca.client.SCAClientFactoryFinder;
3791
3792  import java.io.BufferedReader;
3793  import java.io.Closeable;
3794  import java.io.IOException;
3795  import java.io.InputStream;
3796  import java.io.InputStreamReader;
3797  import java.lang.reflect.Constructor;
3798  import java.net.URI;
3799  import java.net.URL;
3800  import java.util.Properties;
3801
3802  import org.oasisopen.sca.NoSuchDomainException;
3803  import org.oasisopen.sca.ServiceRuntimeException;
3804  import org.oasisopen.sca.client.SCAClientFactory;
3805
3806  /**
3807  * This is a default implementation of an SCAClientFactoryFinder which is
3808  * used to find an implementation of the SCAClientFactory interface.

```

```

3809 *
3810 * @see SCAClientFactoryFinder
3811 * @see SCAClientFactory
3812 *
3813 * @author OASIS Open
3814 */
3815 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
3816
3817     /**
3818      * The name of the System Property used to determine the SPI
3819      * implementation to use for the SCAClientFactory.
3820      */
3821     private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
3822         SCAClientFactory.class.getName();
3823
3824     /**
3825      * The name of the file loaded from the ClassPath to determine
3826      * the SPI implementation to use for the SCAClientFactory.
3827      */
3828     private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
3829         = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
3830
3831     /**
3832      * Public Constructor
3833      */
3834     public SCAClientFactoryFinderImpl() {
3835     }
3836
3837     /**
3838      * Creates an instance of the SCAClientFactorySPI implementation.
3839      * This discovers the SCAClientFactorySPI Implementation and instantiates
3840      * the provider's implementation.
3841      *
3842      * @param properties Properties that may be used when creating a new
3843      * instance of the SCAClient
3844      * @param classLoader ClassLoader that may be used when creating a new
3845      * instance of the SCAClient
3846      * @return new instance of the SCAClientFactory
3847      * @throws ServiceRuntimeException Failed to create SCAClientFactory
3848      * Implementation.
3849      */
3850     public SCAClientFactory find(Properties properties,
3851         ClassLoader classLoader,
3852         URI domainURI )
3853         throws NoSuchDomainException, ServiceRuntimeException {
3854         if (classLoader == null) {
3855             classLoader = getThreadContextClassLoader ();
3856         }
3857         final String factoryImplClassName =
3858             discoverProviderFactoryImplClass(properties, classLoader);
3859         final Class<? extends SCAClientFactory> factoryImplClass
3860             = loadProviderFactoryClass(factoryImplClassName,
3861                 classLoader);
3862         final SCAClientFactory factory =
3863             instantiateSCAClientFactoryClass(factoryImplClass,
3864                 domainURI );
3865         return factory;
3866     }
3867
3868     /**
3869      * Gets the Context ClassLoader for the current Thread.
3870      *
3871      * @return The Context ClassLoader for the current Thread.
3872      */

```

```

3873 private static ClassLoader getThreadContextClassLoader () {
3874     final ClassLoader threadClassLoader =
3875         Thread.currentThread().getContextClassLoader();
3876     return threadClassLoader;
3877 }
3878
3879 /**
3880  * Attempts to discover the class name for the SCAClientFactorySPI
3881  * implementation from the specified Properties, the System Properties
3882  * or the specified ClassLoader.
3883  *
3884  * @return The class name of the SCAClientFactorySPI implementation
3885  * @throws ServiceRuntimeException Failed to find implementation for
3886  * SCAClientFactorySPI.
3887  */
3888 private static String
3889     discoverProviderFactoryImplClass(Properties properties,
3890                                     ClassLoader classLoader)
3891     throws ServiceRuntimeException {
3892     String providerClassName =
3893         checkPropertiesForSPIClassName(properties);
3894     if (providerClassName != null) {
3895         return providerClassName;
3896     }
3897
3898     providerClassName =
3899         checkPropertiesForSPIClassName(System.getProperties());
3900     if (providerClassName != null) {
3901         return providerClassName;
3902     }
3903
3904     providerClassName =
3905         checkMETAINFServicesForSPIClassNamecheckMETAINFServicesForSPIClassName(classLo
3906     ader);
3907     if (providerClassName == null) {
3908         throw new ServiceRuntimeException(
3909             "Failed to find implementation for SCAClientFactory");
3910     }
3911     return providerClassName;
3912 }
3913
3914 /**
3915  * Attempts to find the class name for the SCAClientFactorySPI
3916  * implementation from the specified Properties.
3917  *
3918  * @return The class name for the SCAClientFactorySPI implementation
3919  * or <code>null</code> if not found.
3920  */
3921 private static String
3922     checkPropertiesForSPIClassName(Properties properties) {
3923     if (properties == null) {
3924         return null;
3925     }
3926
3927     final String providerClassName =
3928         properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
3929     if (providerClassName != null && providerClassName.length() > 0) {
3930         return providerClassName;
3931     }
3932
3933     return null;
3934 }
3935
3936

```

```

3937 /**
3938  * Attempts to find the class name for the SCAClientFactorySPI
3939  * implementation from the META-INF/services directory
3940  *
3941  * @return The class name for the SCAClientFactorySPI implementation or
3942  * <code>null</code> if not found.
3943  */
3944 private static String
3945 checkMETAINFServicesForSPIClassNamecheckMETAINFServicesForSPIClassName(ClassLoader cl)
3946 {
3947     final URL url =
3948         cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
3949     if (url == null) {
3950         return null;
3951     }
3952
3953     InputStream in = null;
3954     try {
3955         in = url.openStream();
3956         BufferedReader reader = null;
3957         try {
3958             reader =
3959                 new BufferedReader(new InputStreamReader(in, "UTF-8"));
3960
3961             String line;
3962             while ((line = readNextLine(reader)) != null) {
3963                 if (!line.startsWith("#") && line.length() > 0) {
3964                     return line;
3965                 }
3966             }
3967
3968             return null;
3969         } finally {
3970             closeStream(reader);
3971         }
3972     } catch (IOException ex) {
3973         throw new ServiceRuntimeException(
3974             "Failed to discover SCAClientFactory provider", ex);
3975     } finally {
3976         closeStream(in);
3977     }
3978 }
3979
3980 /**
3981  * Reads the next line from the reader and returns the trimmed version
3982  * of that line
3983  *
3984  * @param reader The reader from which to read the next line
3985  * @return The trimmed next line or <code>null</code> if the end of the
3986  * stream has been reached
3987  * @throws IOException I/O error occurred while reading from Reader
3988  */
3989 private static String readNextLine(BufferedReader reader)
3990     throws IOException {
3991     String line = reader.readLine();
3992     if (line != null) {
3993         line = line.trim();
3994     }
3995     return line;
3996 }
3997
3998 /**
4000

```

```

4001 * Loads the specified SCAClientFactory Implementation class.
4002 *
4003 * @param factoryImplClassName The name of the SCAClientFactory
4004 * Implementation class to load
4005 * @return The specified SCAClientFactory Implementation class
4006 * @throws ServiceRuntimeException Failed to load the SCAClientFactory
4007 * Implementation class
4008 */
4009 private static Class<? extends SCAClientFactory>
4010     loadProviderFactoryClass(String factoryImplClassName,
4011                             ClassLoader classLoader)
4012     throws ServiceRuntimeException {
4013
4014     try {
4015         final Class<?> providerClass =
4016             classLoader.loadClass(factoryImplClassName);
4017         final Class<? extends SCAClientFactory> providerFactoryClass =
4018             providerClass.asSubclass(SCAClientFactory.class);
4019         return providerFactoryClass;
4020     } catch (ClassNotFoundException ex) {
4021         throw new ServiceRuntimeException(
4022             "Failed to load SCAClientFactory implementation class "
4023             + factoryImplClassName, ex);
4024     } catch (ClassCastException ex) {
4025         throw new ServiceRuntimeException(
4026             "Loaded SCAClientFactory implementation class "
4027             + factoryImplClassName
4028             + " is not a subclass of "
4029             + SCAClientFactory.class.getName() , ex);
4030     }
4031 }
4032
4033 /**
4034 * Instantiate an instance of the specified SCAClientFactorySPI
4035 * Implementation class.
4036 *
4037 * @param factoryImplClass The SCAClientFactorySPI Implementation
4038 * class to instantiate.
4039 * @return An instance of the SCAClientFactorySPI Implementation class
4040 * @throws ServiceRuntimeException Failed to instantiate the specified
4041 * specified SCAClientFactorySPI Implementation class
4042 */
4043 private static SCAClientFactory instantiateSCAClientFactoryClass(
4044     Class<? extends SCAClientFactory> factoryImplClass,
4045     URI domainURI)
4046     throws NoSuchDomainException, ServiceRuntimeException {
4047
4048     try {
4049         Constructor<? extends SCAClientFactory> URIConstructor =
4050             factoryImplClass.getConstructor(domainURI.getClass());
4051         SCAClientFactory provider =
4052             URIConstructor.newInstance( domainURI );
4053         return provider;
4054     } catch (Throwable ex) {
4055         throw new ServiceRuntimeException(
4056             "Failed to instantiate SCAClientFactory implementation class "
4057             + factoryImplClass, ex);
4058     }
4059 }
4060
4061 /**
4062 * Utility method for closing Closeable Object.
4063 *
4064 * @param closeable The Object to close.

```

```

4065     */
4066     private static void closeStream(Closeable closeable) {
4067         if (closeable != null) {
4068             try{
4069                 closeable.close();
4070             } catch (IOException ex) {
4071                 throw new ServiceRuntimeException("Failed to close stream",
4072                                                     ex);
4073             }
4074         }
4075     }
4076 }

```

## 4077 B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?

4078 The SCAClient classes and interfaces are designed so that vendors can provide their own  
4079 implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor  
4080 needs to consider in relation to the SCAClient classes and interfaces.

- 4081 • Implement their SCAClientFactory implementation class

4082 Vendors need to provide a subclass of SCAClientFactory that is capable of looking up Services in  
4083 their SCA Runtime. Vendors need to subclass SCAClientFactory and implement the getService()  
4084 method so that it creates reference proxies to services in SCA Domains handled by their SCA  
4085 runtime(s).

- 4086 • Configure the Vendor SCAClientFactory implementation class so that it gets used

4087 Vendors have several options:

4088 Option 1: Set System Property to point to the Vendor's implementation

4089 Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their  
4090 implementation class and use the reference implementation of SCAClientFactoryFinder

4091 Option 2: Provide a META-INF/services file

4092 Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points  
4093 to their implementation class and use the reference implementation of SCAClientFactoryFinder

4094 Option 3: Inject a vendor implementation of the SCAClientFactoryFinder interface into  
4095 SCAClientFactory

4096 Vendors inject an instance of the vendor implementation of SCAClientFactoryFinder into the  
4097 factoryFinder field of the SCAClientFactory abstract class. The reference implementation of  
4098 SCAClientFactoryFinder is not used in this scenario. The vendor implementation of  
4099 SCAClientFactoryFinder can find the vendor implementation(s) of SCAClientFactory by any  
4100 means.

4101

## C. Conformance Items

4102 This section contains a list of conformance items for the SCA-J Common Annotations and APIs  
4103 specification.  
4104

Conformance ID	Description
[JCA20001]	Remotable Services <b>MUST NOT</b> make use of <i>method overloading</i> .
[JCA20002]	the SCA runtime <b>MUST</b> ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
[JCA20003]	within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime <b>MUST</b> only make a single invocation of one business method.
[JCA20004]	Where an implementation is used by a "domain level component", and the implementation is marked "Composite" scope, the SCA runtime <b>MUST</b> ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation.
[JCA20005]	When the implementation class is marked for eager initialization, the SCA runtime <b>MUST</b> create a composite scoped instance when its containing component is started.
[JCA20006]	If a method of an implementation class is marked with the @Init annotation, the SCA runtime <b>MUST</b> call that method when the implementation instance is created.
[JCA20007]	the SCA runtime <b>MAY</b> run multiple threads in a single composite scoped implementation instance object and the SCA runtime <b>MUST NOT</b> perform any synchronization.
[JCA20008]	Where an implementation is marked "Composite" scope and it is used by a component that is nested inside a composite that is used as the implementation of a higher level component, the SCA runtime <b>MUST</b> ensure that all consumers of the component appear to be interacting with a single runtime instance of the implementation. There can be multiple instances of the higher level component, each running on different nodes in a distributed SCA runtime.
[JCA20009]	The SCA runtime <b>MAY</b> use by-reference semantics when passing input parameters, return values or exceptions on calls to remotable services within the same JVM if both the service method implementation and the service proxy used by the client are marked "allows pass by reference".
[JCA20010]	The SCA runtime <b>MUST</b> use by-value semantics when passing input parameters, return values and exceptions on calls to remotable services within the same JVM if the service method implementation is not marked "allows pass by reference" or the service proxy used by the client is not marked "allows pass by reference".
[JCA30001]	The value of the @interface attribute <b>MUST</b> be the fully qualified name of the Java interface class
[JCA30002]	The value of the @callbackInterface attribute <b>MUST</b> be the fully



qualified name of a Java interface used for callbacks

- [JCA30003] 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.
- [JCA30004] The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.
- [JCA30005] The value of the @remotable attribute on the <interface.java/> element does not override the presence of a @Remotable annotation on the interface class and so if the interface class contains a @Remotable annotation and the @remotable attribute has a value of "false", then the SCA Runtime MUST raise an error and MUST NOT run the component concerned.
- [JCA30006] A Java interface referenced by the @interface attribute of an <interface.java/> element MUST NOT contain any of the following SCA Java annotations:  
@AllowsPassByReference, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service.
- [JCA30007] A Java interface referenced by the @callbackInterface attribute of an <interface.java/> element MUST NOT contain any of the following SCA Java annotations:  
@AllowsPassByReference, @Callback, @ComponentName, @Constructor, @Context, @Destroy, @EagerInit, @Init, @Intent, @Property, @Qualifier, @Reference, @Scope, @Service.
- [JCA30009] The SCA Assembly Model specification [ASSEMBLY] defines a number of criteria that need to be satisfied in order for two interfaces to be compatible or have a compatible superset or subset relationship. If these interfaces are both Java interfaces, compatibility also means that every method that is present in both interfaces is defined consistently in both interfaces with respect to the @OneWay annotation, that is, the annotation is either present in both interfaces or absent in both interfaces.
- [JCA30010] If the identified class is annotated with either the JAX-WS @WebService or @WebServiceProvider annotations and the annotation has a non-empty **wsdlLocation** property, then the SCA Runtime MUST act as if an <interface.wsdl/> element is present instead of the <interface.java/> element, with an @interface attribute identifying the portType mapped from the Java interface class and containing @requires and @policySets attribute values equal to the @requires and @policySets attribute values of the <interface.java/> element.
- [JCA40001] The SCA Runtime MUST call a constructor of the component implementation at the start of the Constructing state.
- [JCA40002] The SCA Runtime MUST perform any constructor reference or property injection when it calls the constructor of a component implementation.
- [JCA40003] When the constructor completes successfully, the SCA Runtime MUST transition the component implementation to the Injecting state.

[JCA40004]	If an exception is thrown whilst in the Constructing state, the SCA Runtime MUST transition the component implementation to the Terminated state.
[JCA40005]	When a component implementation instance is in the Injecting state, the SCA Runtime MUST first inject all field and setter properties that are present into the component implementation.
[JCA40006]	When a component implementation instance is in the Injecting state, the SCA Runtime MUST inject all field and setter references that are present into the component implementation, after all the properties have been injected.
[JCA40007]	The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected properties and references are made visible to the component implementation without requiring the component implementation developer to do any specific synchronization.
[JCA40008]	The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation is in the Injecting state.
[JCA40009]	When the injection of properties and references completes successfully, the SCA Runtime MUST transition the component implementation to the Initializing state.
[JCA40010]	If an exception is thrown whilst injecting properties or references, the SCA Runtime MUST transition the component implementation to the Destroying state.
[JCA40011]	When the component implementation enters the Initializing State, the SCA Runtime MUST call the method annotated with @Init on the component implementation, if present.
[JCA40012]	If a component implementation invokes an operation on an injected reference that refers to a target that has not yet been initialized, the SCA Runtime MUST throw a ServiceUnavailableException.
[JCA40013]	The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Initializing state.
[JCA40014]	Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition the component implementation to the Running state.
[JCA40015]	If an exception is thrown whilst initializing, the SCA Runtime MUST transition the component implementation to the Destroying state.
[JCA40016]	The SCA Runtime MUST invoke Service methods on a component implementation instance when the component implementation is in the Running state and a client invokes operations on a service offered by the component.
[JCA40017]	When the component implementation scope ends, the SCA Runtime MUST transition the component implementation to the Destroying state.
[JCA40018]	When a component implementation enters the Destroying state, the SCA Runtime MUST call the method annotated with @Destroy on the

component implementation, if present.

- [JCA40019] If a component implementation invokes an operation on an injected reference that refers to a target that has been destroyed, the SCA Runtime MUST throw an `InvalidServiceException`.
- [JCA40020] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Destroying state.
- [JCA40021] Once the method annotated with `@Destroy` completes successfully, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40022] If an exception is thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the Terminated state.
- [JCA40023] The SCA Runtime MUST NOT invoke Service methods on the component implementation when the component implementation instance is in the Terminated state.
- [JCA40024] If a property or reference is unable to be injected, the SCA Runtime MUST transition the component implementation to the Destroying state.
- [JCA60001] When a bidirectional service is invoked, the SCA runtime MUST inject a callback reference for the invoking service into all fields and setter methods of the service implementation class that are marked with a `@Callback` annotation and typed by the callback interface of the bidirectional service, and the SCA runtime MUST inject null into all other fields and setter methods of the service implementation class that are marked with a `@Callback` annotation.
- [JCA60002] When a non-bidirectional service is invoked, the SCA runtime MUST inject null into all fields and setter methods of the service implementation class that are marked with a `@Callback` annotation.
- [JCA60003] The SCA asynchronous service Java interface mapping of a WSDL request-response operation MUST appear as follows:
- The interface is annotated with the "asyncInvocation" intent.
  - For each service operation in the WSDL, the Java interface contains an operation with
    - a name which is the JAX-WS mapping of the WSDL operation name, with the suffix "Async" added
    - a void return type
    - a set of input parameter(s) which match the JAX-WS mapping of the input parameter(s) of the WSDL operation plus an additional last parameter which is a `ResponseDispatch` object typed by the JAX-WS Response Bean mapping of the output parameter(s) of the WSDL operation, where `ResponseDispatch` is the type defined in the SCA Java Common Annotations and APIs specification.
- [JCA60004] An SCA Runtime MUST support the use of the SCA asynchronous service interface for the interface of an SCA service.
- [JCA60005] If the SCA asynchronous service interface `ResponseDispatch` `handleResponse` method is invoked more than once through either its

sendResponse or its sendFault method, the SCA runtime MUST throw an IllegalStateException.

[JCA60006]

For the purposes of matching interfaces (when wiring between a reference and a service, or when using an implementation class by a component), an interface which has one or more methods which follow the SCA asynchronous service pattern MUST be treated as if those methods are mapped as the equivalent synchronous methods, as follows:

Asynchronous service methods are characterized by:

- void return type
- a method name with the suffix "Async"
- a last input parameter with a type of ResponseDispatch<X>
- annotation with the asyncInvocation intent
- possible annotation with the @AsyncFault annotation

The mapping of each such method is as if the method had the return type "X", the method name without the suffix "Async" and all the input parameters except the last parameter of the type ResponseDispatch<X>, plus the list of exceptions contained in the @AsyncFault annotation.

[JCA70001]

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

[JCA70002]

Intent annotations MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with @Reference or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class field that is not either annotated with @Reference or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with @Reference

[JCA70003]

Where multiple intent annotations (general or specific) are applied to the same Java element, the SCA runtime MUST compute the combined intents for the Java element by merging the intents from all intent annotations on the Java element according to the SCA Policy Framework [POLICY] rules for merging intents at the same hierarchy level.

[JCA70004]

If intent annotations are specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective intents for the method by merging the combined intents from the method with the combined intents for the interface according to the SCA Policy Framework [POLICY] rules for merging intents within a structural hierarchy, with the method at the lower level and the interface at the higher level.

[JCA70005]

The @PolicySets annotation MUST NOT be applied to the following:

- A method of a service implementation class, except for a setter method that is either annotated with @Reference or introspected

as an SCA reference according to the rules in the appropriate Component Implementation specification

- A service implementation class field that is not either annotated with @Reference or introspected as an SCA reference according to the rules in the appropriate Component Implementation specification
- A service implementation class constructor parameter that is not annotated with @Reference

[JCA70006]	If the @PolicySets annotation is specified on both an interface method and the method's declaring interface, the SCA runtime MUST compute the effective policy sets for the method by merging the policy sets from the method with the policy sets from the interface.
[JCA80001]	The ComponentContext.getService method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..n or 1..n.
[JCA80002]	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.
[JCA80003]	When invoked during the execution of a service operation, the RequestContext.getServiceReference method MUST return a ServiceReference that represents the service that was invoked.
[JCA80004]	The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the reference named by the referenceName parameter has multiplicity greater than one.
[JCA80005]	The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the reference named by the referenceName parameter does not have an interface of the type defined by the businessInterface parameter.
[JCA80006]	The ComponentContext.getServiceReference method MUST throw an IllegalArgumentException if the component does not have a reference with the name provided in the referenceName parameter.
[JCA80007][JCA80007]	The ComponentContext.getServiceReference method MUST return null if the multiplicity of the reference named by the referenceName parameter is 0..1 and the reference has no target service configured.
[JCA80008]	The ComponentContext.getURI method MUST return the <u>absolute structural</u> URI of the component in the SCA Domain.
[JCA80009]	The ComponentContext.getService method MUST return the proxy object implementing the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
[JCA80010]	The ComponentContext.getService method MUST return null if the multiplicity of the reference named by the referenceName parameter is 0..1 and the reference has no target service configured.
[JCA80011]	The ComponentContext.getService method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.

[JCA80012]	The ComponentContext.getService method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
[JCA80013]	The ComponentContext.getServiceReference method MUST return a ServiceReference object typed by the interface provided by the businessInterface parameter, for the reference named by the referenceName parameter with the interface defined by the businessInterface parameter when that reference has a target service configured.
[JCA80014]	The ComponentContext.getServices method MUST return a collection containing one proxy object implementing the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
[JCA80015]	The ComponentContext.getServices method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
[JCA80016]	The ComponentContext.getServices method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
[JCA80017]	The ComponentContext.getServices method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
The ComponentContext.getServices method MUST throw an IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.	[JCA80018]
[JCA80019]	The ComponentContext.getServiceReferences method MUST return a collection containing one ServiceReference object typed by the interface provided by the businessInterface parameter for each of the target services configured on the reference identified by the referenceName parameter.
[JCA80020]	The ComponentContext.getServiceReferences method MUST return an empty collection if the service reference with the name supplied in the referenceName parameter is not wired to any target services.
[JCA80021]	The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..1 or 1..1.
[JCA80022]	The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the component does not have a reference with the name supplied in the referenceName parameter.
[JCA80023]	The ComponentContext.getServiceReferences method MUST throw an

	IllegalArgumentException if the service reference with the name supplied in the referenceName does not have an interface compatible with the interface supplied in the businessInterface parameter.
[JCA80024]	The ComponentContext.createSelfReference method MUST return a ServiceReference object typed by the interface defined by the businessInterface parameter for one of the services of the invoking component which has the interface defined by the businessInterface parameter.
[JCA80025]	The ComponentContext.getServiceReferences method MUST throw an IllegalArgumentException if the component does not have a service which implements the interface identified by the businessInterface parameter.
[JCA80026]	The ComponentContext.createSelfReference method MUST return a ServiceReference object typed by the interface defined by the businessInterface parameter for the service identified by the serviceName of the invoking component and which has the interface defined by the businessInterface parameter.
[JCA80027]	The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the component does not have a service with the name identified by the serviceName parameter.
[JCA80028]	The ComponentContext.createSelfReference method MUST throw an IllegalArgumentException if the component service with the name identified by the serviceName parameter does not implement a business interface which is compatible with the supplied businessInterface parameter.
[JCA80029]	The ComponentContext.getProperty method MUST return an object of the type identified by the type parameter containing the value specified in the component configuration for the property named by the propertyName parameter or null if no value is specified in the configuration.
[JCA80030]	The ComponentContext.getProperty method MUST throw an IllegalArgumentException if the component does not have a property with the name identified by the propertyName parameter.
[JCA80031]	The ComponentContext.getProperty method MUST throw an IllegalArgumentException if the component property with the name identified by the propertyName parameter does not have a type which is compatible with the supplied type parameter.
[JCA80032]	The ComponentContext.cast method MUST return a ServiceReference object which is typed by the same business interface as specified by the reference proxy object supplied in the target parameter.
[JCA80033]	The ComponentContext.cast method MUST throw an IllegalArgumentException if the supplied target parameter is not an SCA reference proxy object.
[JCA80034]	The RequestContext.getSecuritySubject method MUST return the JAAS subject of the current request, or null if there is no subject or null if the method is invoked from code not processing a service request or callback request.
[JCA80035]	The RequestContext.getServiceName method MUST return the name of the service for which an operation is being processed, or null if

invoked from a thread that is not processing a service operation or a callback operation.

- [JCA80036] The RequestContext.getCallbackReference method MUST return a ServiceReference object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80037] The RequestContext.getCallback method MUST return a reference proxy object typed by the interface of the callback supplied by the client of the invoked service, or null if either the invoked service is not bidirectional or if the method is invoked from a thread that is not processing a service operation.
- [JCA80038] When invoked during the execution of a callback operation, the RequestContext.getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.
- [JCA80039] When invoked from a thread not involved in the execution of either a service operation or of a callback operation, the RequestContext.getServiceReference method MUST return null.
- [JCA80040] The ServiceReference.getService method MUST return a reference proxy object which can be used to invoke operations on the target service of the reference and which is typed with the business interface of the reference.
- [JCA80041] The ServiceReference.getBusinessInterface method MUST return a Class object representing the business interface of the reference.
- [JCA80042] The SCAClientFactory.newInstance( URI ) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80043] The SCAClientFactory.newInstance( URI ) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80044] The SCAClientFactory.newInstance( Properties, URI ) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80045] The SCAClientFactory.newInstance( Properties, URI ) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80046] The SCAClientFactory.newInstance( Classloader, URI ) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80047] The SCAClientFactory.newInstance( Classloader, URI ) method MUST throw a NoSuchDomainException if the domainURI parameter does not identify a valid SCA Domain.
- [JCA80048] The SCAClientFactory.newInstance( Properties, Classloader, URI ) method MUST return an object which implements the SCAClientFactory class for the SCA Domain identified by the domainURI parameter.
- [JCA80049] The SCAClientFactory.newInstance( Properties, Classloader, URI )



	MUST throw a <code>NoSuchDomainException</code> if the <code>domainURI</code> parameter does not identify a valid SCA Domain.
[JCA80050]	The <code>SCAClientFactory.getService</code> method MUST return a proxy object which implements the business interface defined by the <code>interfaze</code> parameter and which can be used to invoke operations on the service identified by the <code>serviceURI</code> parameter.
[JCA80051]	The <code>SCAClientFactory.getService</code> method MUST throw a <code>NoSuchServiceException</code> if a service with the relative URI <code>serviceURI</code> and a business interface which matches <code>interfaze</code> cannot be found in the SCA Domain targeted by the <code>SCAClient</code> object.
[JCA80052]	The <code>SCAClientFactory.getService</code> method MUST throw a <code>NoSuchServiceException</code> if the <code>domainURI</code> of the <code>SCAClientFactory</code> does not identify a valid SCA Domain.
[JCA80053]	The <code>SCAClientFactory.getDomainURI</code> method MUST return the SCA Domain URI of the Domain associated with the <code>SCAClientFactory</code> object.
[JCA80054]	The <code>SCAClientFactory.getDomainURI</code> method MUST throw a <b><code>NoSuchServiceException</code></b> if the <code>domainURI</code> of the <code>SCAClientFactory</code> does not identify a valid SCA Domain.
The implementation of the <code>SCAClientFactoryFinder.find</code> method MUST return an object which is an implementation of the <code>SCAClientFactory</code> interface, for the SCA Domain represented by the <code>doaminURI</code> parameter, using the supplied properties and classloader.	The implementation of the <code>SCAClientFactoryFinder.find</code> method MUST return an object which is an implementation of the <code>SCAClientFactory</code> interface, for the SCA Domain represented by the <code>doaminURI</code> parameter, using the supplied properties and classloader.
[JCA80055]	
[JCA80056]	The implementation of the <code>SCAClientFactoryFinder.find</code> method MUST throw a <code>ServiceRuntimeException</code> if the <code>SCAClientFactory</code> implementation could not be found.
[JCA50057]	The <code>ResponseDispatch.sendResponse()</code> method MUST send the response message to the client of an asynchronous service.
[JCA80058]	The <code>ResponseDispatch.sendResponse()</code> method MUST throw an <code>InvalidStateException</code> if either the <code>sendResponse</code> method or the <code>sendFault</code> method has already been called once.
[JCA80059]	The <code>ResponseDispatch.sendFault()</code> method MUST send the supplied fault to the client of an asynchronous service.
[JCA80060]	The <code>ResponseDispatch.sendFault()</code> method MUST throw an <code>InvalidStateException</code> if either the <code>sendResponse</code> method or the <code>sendFault</code> method has already been called once.
[JCA90001]	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.

- [JCA90003] 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.
- [JCA90004] A method annotated with `@Destroy` can have any access modifier and MUST have a void return type and no arguments.
- [JCA90005] 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.
- [JCA90007] When marked for eager initialization with an `@EagerInit` annotation, the composite scoped instance MUST be created when its containing component is started.
- [JCA90008] A method marked with the `@Init` annotation can have any access modifier and MUST have a void return type and no arguments.
- [JCA90009] If there is a method annotated with `@Init` that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.
- [JCA90011] The `@Property` annotation MUST NOT be used on a class field that is declared as final.
- [JCA90013] 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.
- [JCA90014] For a `@Property` annotation applied to a constructor parameter, the required attribute MUST NOT have the value false.
- [JCA90015] The `@Qualifier` annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
- [JCA90016] The `@Reference` annotation MUST NOT be used on a class field that is declared as final.
- [JCA90018] For a `@Reference` annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.
- [JCA90019] For a `@Reference` annotation applied to a constructor parameter, the required attribute MUST have the value true.
- [JCA90020] 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.
- [JCA90021] 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.

- [JCA90022] 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).
- [JCA90023] 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).
- [JCA90024] 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.
- [JCA90025] 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.
- [JCA90026] 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.
- [JCA90027] If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException.
- [JCA90028] 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.
- [JCA90029] If the target service of the reference is changed, the reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
- [JCA90030] 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.
- [JCA90031] If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException when an operation is invoked on the ServiceReference.
- [JCA90032] If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.
- [JCA90033] If the target service of a ServiceReference is changed, the reference MUST either continue to work or throw an InvalidServiceException when it is invoked.
- [JCA90034] 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.
- [JCA90035] 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`.

[JCA90036]

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.

[JCA90037]

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.

[JCA90038]

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

[JCA90039]

A reinjected array or `Collection` for a reference MUST NOT be the same array or `Collection` object previously injected to the component.

[JCA90040]

A remotable service can be published externally as a service and MUST be translatable into a WSDL `portType`.

[JCA90041]

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

[JCA90042]

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.

[JCA90045]

If a component implementation has two services with the same Java simple name, the `names` attribute of the `@Service` annotation MUST be specified.

[JCA90046]

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.

[JCA90047]

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 SCA runtime MUST introspect the component type of the implementation with a `<property/>` element with a `@many` attribute set to true, otherwise `@many` MUST be set to false.

[JCA90050]

The number of Strings in the `names` attribute array of the `@Service` annotation MUST match the number of elements in the `value` attribute array.

[JCA90052]

The `@AllowsPassByReference` annotation MUST only annotate the following locations:

- a service implementation class
- an individual method of a remotable service implementation
- an individual reference which uses a remotable interface, where the reference is a field, a setter method, or a constructor parameter

[JCA90053]	The @Remotable annotation is valid only on a Java interface, a Java class, a field, a setter method, or a constructor parameter. It MUST NOT appear anywhere else.
[JCA90054]	When used to annotate a method or a field of an implementation class for injection of a callback object, the type of the method or field MUST be the callback interface of at least one bidirectional service offered by the implementation class.
[JCA90055]	A method annotated with @OneWay MUST have a void return type and MUST NOT have declared checked exceptions.
[JCA90056]	When a method of a Java interface is annotated with @OneWay, the SCA runtime MUST ensure that all invocations of that method are executed in a non-blocking fashion, as described in the section on Asynchronous Programming.
[JCA90057]	The @Callback annotation MUST NOT appear on a setter method or a field of a Java implementation class that has COMPOSITE scope.
[JCA90058]	When used to annotate a setter method or a field of an implementation class for injection of a callback object, the SCA runtime MUST inject a callback reference proxy into that method or field when the Java class is initialized, if the component is invoked via a service which has a callback interface and where the type of the setter method or field corresponds to the type of the callback interface.
[JCA90060]	The value of each element in the @Service names array MUST be unique amongst all the other element values in the array.
[JCA90061]	When the Java type of a field, setter method or constructor parameter with the @Property annotation is a primitive type or a JAXB annotated class, the SCA runtime MUST convert a property value specified by an SCA component definition into an instance of the Java type as defined by the XML to Java mapping in the JAXB specification [JAXB] with XML schema validation enabled.
[JCA100001]	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.
[JCA100002]	The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.
[JCA100003]	For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.
[JCA100004]	SCA runtimes MUST support the JAXB 2.1 mapping from XML Schema to Java and from Java to XML Schema.
[JCA100005]	SCA runtimes MAY support the SDO 2.1 mapping from XML schema types to Java and from Java to XML Schema.
[JCA100006]	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.
[JCA100007]	For SCA reference interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the additional

client-side asynchronous polling and callback methods defined by JAX-WS.

- [JCA100008] If the additional client-side asynchronous polling and callback 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.
- [JCA100009] SCA runtimes MUST support the use of the JAX-WS client asynchronous model.
- [JCA100010] For SCA service interfaces defined using interface.java, the SCA runtime MUST support a Java interface which contains the server-side asynchronous methods defined by SCA.
- [JCA100011] An SCA runtime MUST apply the JAX-WS annotations as described in Table 11-1 and Table 11-2 when introspecting a Java class or interface class.
- [JCA100012] A Java interface or class annotated with @WebService MUST be treated as if annotated with the SCA @Remotable annotation
- [JCA100013] A Java class annotated with the @WebService annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition instead of the annotated Java class.
- [JCA100014] A Java class annotated with the @WebService annotation with its endpointInterface attribute set MUST have its interface defined by the referenced interface instead of annotated Java class.
- [JCA100015] A Java class or interface containing an @WebParam annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100016] A Java class or interface containing an @WebResult annotation with its header attribute set to "true" MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100017] A Java class containing an @ServiceMode annotation MUST be treated as if the SOAP intent is applied to the Java class.
- [JCA100018] An interface or class annotated with @WebServiceClient MUST NOT be used to define an SCA interface.
- [JCA100019] A class annotated with @WebServiceProvider MUST be treated as if annotated with the SCA @Remotable annotation.
- [JCA100020] A Java class annotated with the @WebServiceProvider annotation with its wsdlLocation attribute set MUST have its interface defined by the referenced WSDL definition is used instead of the annotated Java class.
- [JCA100021] A Java class or interface containing an @SOAPBinding annotation MUST be treated as if the SOAP intent is applied to the Java class or interface.
- [JCA100022] SCA runtimes MUST support the JAX-WS 2.1 mappings from WSDL to Java and from Java to WSDL.
- [JCA100023] The value of the name attribute of the @WebService annotation, if present, MUST be used to define the name of an SCA service when

	there is no @Service annotation present in the SCA component implementation.
[JCA100024]	For a Java method annotated with the @WebMethod annotation with the operationName set, an SCA runtime MUST use the value of the operationName attribute as the SCA operation name.
[JCA100025]	An SCA runtime MUST NOT include a Java method annotated with the @WebMethod annotation with the exclude attribute set to true in an SCA interface.
[JCA100026]	For a Java parameter annotated with the @WebParam annotation with the mode attribute set, an SCA runtime MUST apply the value of the mode attribute when comparing interfaces.
The value of the name attribute of the @WebService annotation, if present, MUST be used to define the name of an SCA service when the @Service annotation is present without the names attribute and indicates that the Java interface or class annotated with the @WebService annotation defines an SCA service interface. [JCA100028]	The value of the name attribute of the @WebService annotation, if present, MUST be used to define the name of an SCA service when the @Service annotation is present without the names attribute and indicates that the Java interface or class annotated with the @WebService annotation defines an SCA service interface.

4106  
4107  
4108  
4109

## D. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

**Participants:**

Participant Name	Affiliation
Bryan Aupperle	IBM
Ron Barack	SAP AG*
<u>Mirza Begg</u>	<u>Individual</u>
Michael Beisiegel	IBM
Henning Blohm	SAP AG*
David Booz	IBM
Martin Chapman	Oracle Corporation
Graham Charters	IBM
Shih-Chang Chen	Oracle Corporation
Chris Cheng	Primeton Technologies, Inc.
Vamsavardhana Reddy Chillakuru	IBM
Roberto Chinnici	Sun Microsystems
Pyungkuk Cho	Oracle Corporation
Eric Clairambault	IBM
Mark Combella	Avaya, Inc.
Jean-Sebastien Delfino	IBM
<u>Derek Dougans</u>	<u>Individual</u>
Mike Edwards	IBM
<u>Ant Elder</u>	<u>IBM</u>
Raymond Feng	IBM
Bo Ji	Primeton Technologies, Inc.
Uday Joshi	Oracle Corporation
Anish Karmarkar	Oracle Corporation
<u>Khanderao Kand</u>	<u>Oracle Corporation</u>
Michael Keith	Oracle Corporation
Rainer Kerth	SAP AG*
Meeraj Kunumpurath	Individual
Simon Laws	IBM
Yang Lei	IBM
Mark Little	Red Hat
Ashok Malhotra	Oracle Corporation
Jim Marino	Individual
Jeff Mischkinsky	Oracle Corporation
Sriram Narasimhan	TIBCO Software Inc.
Simon Nash	Individual
Sanjay Patil	SAP AG*
Plamen Pavlov	SAP AG*
Peter Peshev	SAP AG*
Ramkumar Ramalingam	IBM
Luciano Resende	IBM



Michael Rowley  
Vladimir Savchenko  
Pradeep Simha  
Raghav Srinivasan  
Scott Vorthmann  
Feng Wang

Robin Paul Yang

Active Endpoints, Inc.  
SAP AG\*  
TIBCO Software Inc.  
Oracle Corporation  
TIBCO Software Inc.  
Primeton Technologies, Inc.  
~~Primeton Technologies,~~  
~~Inc.~~ Changfeng Open  
Standards Platform Software

4110

## E. Revision History

4111 [optional; should not be included in OASIS Standards]

4112

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 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
WD05	2008-10-03	Anish Karmarkar	* Fixed the resolution of issue 37 but re-adding the sentence: "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 > 1 * minor ed changes
cd01-rev1	2008-12-11	Anish Karmarkar	* Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup.

			All changes accepted. All comments removed.
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	2009-03-16	Mike Edwards	Issue 104 - RFC2119 work and formal marking of all normative statements - all sections - Completion of Appendix B (list of all normative statements) Accept all changes
cd02-rev4	2009-03-20	Mike Edwards	Editorially removed sentence about componentType side files in Section1 Editorially changed package name to org.oasisopen from org.osoa in lines 291, 292 Issue 6 - add Section 2.3, modify section 9.1 Issue 30 - Section 2.2.2 Issue 76 - Section 6.2.4 Issue 27 - Section 7.6.2, 7.6.2.1 Issue 77 - Section 1.2 Issue 102 - Section 9.21 Issue 123 - conersations removed Issue 65 - Added a new Section 4 ** Causes renumbering of later sections ** ** NB new numbering is used below ** Issue 119 - Added a new section 12 Issue 125 - Section 3.1 Issue 130 - (new number) Section 8.6.2.1 Issue 132 - Section 1 Issue 133 - Section 10.15, Section 10.17 Issue 134 - Section 10.3, Section 10.18 Issue 135 - Section 10.21 Issue 138 - Section 11 Issue 141 - Section 9.1 Issue 142 - Section 10.17.1
cd02-rev5	2009-04-20	Mike Edwards	Issue 154 - Appendix A Issue 129 - Section 8.3.1.1
cd02-rev6	2009-04-28	Mike Edwards	Issue 148 - Section 3 Issue 98 - Section 8

cd02-rev7	2009-04-30	Mike Edwards	Editorial cleanup throughout the spec
cd02-rev8	2009-05-01	Mike Edwards	Further extensive editorial cleanup throughout the spec Issue 160 - Section 8.6.2 & 8.6.2.1 removed
cd02-rev8a	2009-05-03	Simon Nash	Minor editorial cleanup
cd03	2009-05-04	Anish Karmarkar	Updated references and front page clean up
cd03-rev1	2009-09-15	David Booz	Applied Issues: 1,13,125,131,156,157,158,159,161,165,172,177
cd03-rev2	2010-01-19	David Booz	Updated to current Assembly namespace Applied issues: 127,155,168,181,184,185,187,189,190,194
cd03-rev3	2010-02-01	Mike Edwards	Applied issue 54. Editorial updates to code samples.
cd03-rev4	2010-02-05	Bryan Aupperle, Dave Booz	Editorial update for OASIS formatting
CD04	2010-02-06	Dave Booz	Editorial updates for Committee Draft 04 All changes accepted
<a href="#">CD04-rev1</a>	<a href="#">2010-07-13</a>	<a href="#">Dave Booz</a>	<a href="#">Applied issues 199, 200</a>
<a href="#">CD04-rev2</a>	<a href="#">2010-10-19</a>	<a href="#">Dave Booz</a>	<a href="#">Applied issues 201,212,213</a>
<a href="#">CSD04-rev3</a>	<a href="#">2010-11-05</a>	<a href="#">Dave Booz</a>	<a href="#">Applied issue 216, ed. updates for CSD vote</a>

4113