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

Committee Draft 03 / Public Review Draft 01

4 May 2009

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.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/sca-j/.

Notices

Copyright © OASIS® 2005, 2009. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1 Introduction

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]	SCA Assembly Model Specification Version 1.1, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd03.pdf
[JAVA_CI]	SCA POJO Component Implementation Specification Version 1.1 http://docs.oasis-open.org/opencsa/sca-j/sca-javaci-1.1-spec-cd01.pdf
[SDO]	SDO 2.1 Specification, http://www.osoa.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]	SCA Policy Framework Version 1.1, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd02.pdf
[JSR-250]	Common Annotations for the Java Platform specification (JSR-250), http://www.jcp.org/en/jsr/detail?id=250
[JAX-WS]	JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224
[JAVABEANS]	JavaBeans 1.01 Specification, http://java.sun.com/javase/technologies/desktop/javabeans/api/

44	**[JAAS]**	Java Authentication and Authorization Service Reference Guide
45		http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.
46		html

1.3 Non-Normative References

| 48 | **[EBNF-Syntax]** | Extended BNF syntax format used for formal grammar of constructs |
| 49 | | http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation |

2 Implementation Metadata

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

2.1 Service Metadata

2.1.1 @Service

The **@Service annotation** is used on a Java class to specify the interfaces of the services 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. Remotable services are intended to be used for **coarse grained** services, and the parameters are passed **by-value**. Remotable Services MUST NOT make use of **method overloading**. [JCA20001]

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

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

2.1.3 Java Semantics of a Local Service

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

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

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

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

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

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

2.1.4 @Reference

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

2.1.5 @Property

Implementations can be configured with data values through the use of properties, as defined in the SCA Assembly 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.

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

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

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

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

2.2.1 Stateless Scope

For stateless scope components, there is no implied correlation between implementation instances used to dispatch service requests.

135	The concurrency model for the stateless scope is single threaded. This means that the SCA
136	runtime MUST ensure that a stateless scoped implementation instance object is only ever
137	dispatched on one thread at any one time. [JCA20002] In addition, within the SCA lifecycle of a
138	stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of
139	one business method. [JCA20003] Note that the SCA lifecycle might not correspond to the Java
140	object lifecycle due to runtime techniques such as pooling.

2.2.2 Composite Scope

| 142 | The meaning of "composite scope" is defined in relation to the composite containing the |
| 143 | component. |

| 144 | It is important to distinguish between different uses of a composite, where these uses affect the |
| 145 | numbers of instances of components within the composite. There are 2 cases: |

| 146 | a) | Where the composite containing the component using the Java implementation is the SCA |
| 147 | | Domain (i.e. a deployment composite declares the component using the implementation) |

148	b)	Where the composite containing the component using the Java implementation is itself used
149		as the implementation of a higher level component (any level of nesting is possible, but the
150		component is NOT at the Domain level)

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

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

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

| 162 | The lifetime of the containing composite is defined as the time it becomes active in the runtime to |
| 163 | the time it is deactivated, either normally or abnormally. |

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

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

2.3 @AllowsPassByReference

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

| 183 | The cost of such copying can be very high relative to the cost of making a local call, especially if |
| 184 | the data being passed is large. Also, in many cases this copying is not needed if the |

185 implementation observes certain conventions for how input parameters, return values and
186 exceptions are used. The @AllowsPassByReference annotation allows service method
187 implementations and client references to be marked as "allows pass by reference" to indicate that
188 they use input parameters, return values and exceptions in a manner that allows the SCA runtime
189 to avoid the cost of copying mutable objects when a remotable service is called locally within the
190 same JVM.

2.3.1 Marking Services and References as "allows pass by reference"

191

192 Marking a service method implementation as "allows pass by reference" asserts that the method
193 implementation observes the following restrictions:

194 • Method execution will not modify any input parameter before the method returns.

195 • The service implementation will not retain a reference to any mutable input parameter,
196 mutable return value or mutable exception after the method returns.

197 • The method will observe "allows pass by value" client semantics (see below) for any
198 callbacks that it makes.

199 See section "@AllowsPassByReference" for details of how the @AllowsPassByReference annotation
200 is used to mark a service method implementation as "allows pass by reference".

201 Marking a client reference as "allows pass by reference" asserts that method calls through the
202 reference observe the following restrictions:

203 • The client implementation will not modify any of the method's input parameters before
204 the method returns. Such modifications might occur in callbacks or separate client
205 threads.

206 • If the method is one-way, the client implementation will not modify any of the method's
207 input parameters at any time after calling the method. This is because one-way method
208 calls return immediately without waiting for the service method to complete.

209 See section "Applying "allows pass by reference" to Service Proxies" for details of how the
210 @AllowsPassByReference annotation is used to mark a client reference as "allows pass by
211 reference".

2.3.2 Applying "allows pass by reference" to Service Proxies

212

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

216 • It is injected into a reference or callback reference that is marked "allows pass by
217 reference".

218 • It is obtained by calling ComponentContext.getService() or
219 ComponentContext.getServices() with the name of a reference that is marked "allows
220 pass by reference".

221 • It is obtained by calling RequestContext.getCallback() from a service implementation that
222 is marked "allows pass by reference".

223 • It is obtained by calling ServiceReference.getService() on a service reference that is
224 marked "allows pass by reference" (see definition below).

225 A service reference for a remotable service call is marked "allows pass by reference" if and only if
226 any of the following applies:

227 • It is injected into a reference or callback reference that is marked "allows pass by
228 reference".

229 • It is obtained by calling ComponentContext.getServiceReference() or
230 ComponentContext.getServiceReferences() with the name of a reference that is marked
231 "allows pass by reference".

232   • It is obtained by calling RequestContext.getCallbackReference() from a service
233   implementation that is marked "allows pass by reference".

234   • It is obtained by calling ComponentContext.cast() on a proxy that is marked "allows pass
235   by reference".

2.3.3 Using "allows pass by reference" to Optimize Remotable Calls

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

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

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

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

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

The interface.java element has the following attributes:

- ***interface : NCName (1..1)*** – the Java interface class to use for the service interface. The value of the @interface attribute MUST be the fully qualified name of the Java interface class [JCA30001]

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

- ***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. 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. 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]

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

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

289   Here, the Java interface is defined in the Java class file
290   ***./services/stockquote/StockQuoteService.class***, where the root directory is defined by the
291   contribution in which the interface exists. Similarly, the callback interface is defined in the Java
292   class file ***./services/stockquote/StockQuoteServiceCallback.class***.

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

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

303 3.2 @Remotable

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

308 3.3 @Callback

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

313 3.4 SCA Java Annotations for Interface Classes

314 A Java implementation class referenced by the @interface or the @callbackInterface attribute of an
315 element MUST NOT contain the following SCA Java annotations:

316   @Intent, @Qualifier. [JCA30008]

317 A Java interface referenced by the @interface attribute of an element MUST NOT contain any of the
318 following SCA Java annotations:

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

321 A Java interface referenced by the @callbackInterface attribute of an element MUST NOT contain
322 any of the following SCA Java annotations:

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

325

4 SCA Component Implementation Lifecycle

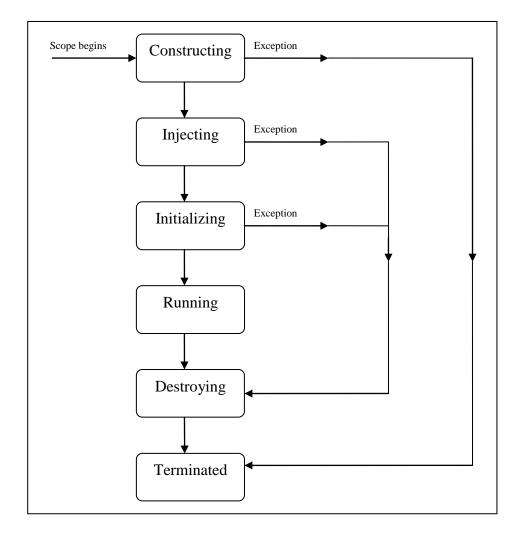This section describes the lifecycle of an SCA component implementation.

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:

1. **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.

2. **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.

3. **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

365 the properties have been injected. [JCA40006] The order in which the references are injected is
366 unspecified.

367 The SCA Runtime MUST ensure that the correct synchronization model is used so that all injected
368 properties and references are made visible to the component implementation without requiring the
369 component implementation developer to do any specific synchronization. [JCA40007]

370 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
371 component implementation is in the Injecting state. [JCA40008]

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

374 When the injection of properties and references completes successfully, the SCA Runtime MUST
375 transition the component implementation to the Initializing state. [JCA40009] If an exception is
376 thrown whilst injecting properties or references, the SCA Runtime MUST transition the component
377 implementation to the Destroying state. [JCA40010]

4.2.3 Initializing State

379 When the component implementation enters the Initializing State, the SCA Runtime MUST call the
380 method annotated with @Init on the component implementation, if present. [JCA40011]

381 The component implementation can invoke operations on any injected references when it is in the
382 Initializing state. However, depending on the order in which the component implementations are
383 initialized, the target of the injected reference might not be available since it has not yet been
384 initialized. If a component implementation invokes an operation on an injected reference that
385 refers to a target that has not yet been initialized, the SCA Runtime MUST throw a
386 ServiceUnavailableException. [JCA40012]

387 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
388 component implementation instance is in the Initializing state. [JCA40013]

389 Once the method annotated with @Init completes successfully, the SCA Runtime MUST transition
390 the component implementation to the Running state. [JCA40014] If an exception is thrown whilst
391 initializing, the SCA Runtime MUST transition the component implementation to the Destroying
392 state. [JCA40015]

4.2.4 Running State

394 The SCA Runtime MUST invoke Service methods on a component implementation instance when
395 the component implementation is in the Running state and a client invokes operations on a service
396 offered by the component. [JCA40016]

397 The component implementation can invoke operations on any injected references when the
398 component implementation instance is in the Running state.

399 When the component implementation scope ends, the SCA Runtime MUST transition the
400 component implementation to the Destroying state. [JCA40017]

4.2.5 Destroying State

402 When a component implementation enters the Destroying state, the SCA Runtime MUST call the
403 method annotated with @Destroy on the component implementation, if present. [JCA40018]

404 The component implementation can invoke operations on any injected references when it is in the
405 Destroying state. However, depending on the order in which the component implementations are
406 destroyed, the target of the injected reference might no longer be available since it has been
407 destroyed. If a component implementation invokes an operation on an injected reference that
408 refers to a target that has been destroyed, the SCA Runtime MUST throw an
409 InvalidServiceException. [JCA40019]

410 The SCA Runtime MUST NOT invoke Service methods on the component implementation when the
411 component implementation instance is in the Destroying state. [JCA40020]

412   Once the method annotated with @Destroy completes successfully, the SCA Runtime MUST
413   transition the component implementation to the Terminated state. [JCA40021] If an exception is
414   thrown whilst destroying, the SCA Runtime MUST transition the component implementation to the
415   Terminated state. [JCA40022]

4.2.6 Terminated State

417   The lifecycle of the SCA Component has ended.

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

5 Client API

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

5.1 Accessing Services from an SCA Component

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

5.1.1 Using the Component Context API

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

5.2 Accessing Services from non-SCA Component Implementations

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

5.2.1 ComponentContext

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

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

440

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

6 Error Handling

445

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

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

449 SCA runtime exceptions are raised by the SCA runtime and signal problems in management of
450 component execution or problems interacting with remote services. The SCA runtime exceptions
451 are 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.

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

For a Java client to make a non-blocking call to methods that either return values or 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.

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

```
package somepackage;
import org.oasisopen.sca.annotation.Callback;
import org.oasisopen.sca.annotation.Remotable;

@Remotable
@Callback(QuotationCallback.class)
public interface Quotation {h
    double requestQuotation(String productCode, int quantity);
}

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

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

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

```
@Callback
protected QuotationCallback callback;

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

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

```
public class ClientImpl implements ClientService, QuotationCallback {

    private QuotationService myService;
```

```
551        @Reference
552        public void setMyService(QuotationService service) {
553            myService = service;
554        }
555
556        public void aClientMethod() {
557            ...
558            double quote = myService.requestQuotation("AB123", 2000);
559            ...
560        }
561
562        public String getState() {
563            return "TX";
564        }
565        public String getZipCode() {
566            return "78746";
567        }
568        public String getCreditRating() {
569            return "AA";
570        }
571    }
572
```

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

7.2.2 Callback Instance Management

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

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

7.2.3 Implementing Multiple Bidirectional Interfaces

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

```
596
597    @Callback
598    protected MyService1Callback callback1;
599
600    @Callback
601    protected MyService2Callback callback2;
602
```

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

7.2.4 Accessing Callbacks

606 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to
607 a Callback instance by annotating a field or method of type **ServiceReference** with the
608 **@Callback** annotation.
609
610 A reference implementing the callback service interface can be obtained using
611 `ServiceReference.getService()`.

612 The following example fragments come from a service implementation that uses the callback API:
613

```
614  @Callback
615  protected ServiceReference<MyCallback> callback;
616
617  public void someMethod() {
618
619      MyCallback myCallback = callback.getService();    …
620
621      myCallback.receiveResult(theResult);
622  }
623
```

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

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

```
630  @Context
631  ComponentContext context;
632
633  public void someMethod() {
634
635      MyCallback myCallback =
636              context.getRequestContext().getCallback();
637
638          …
639
640      myCallback.receiveResult(theResult);
641  }
642
```

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

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.

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 as follows:

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

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

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

```
public static final String SCA_PREFIX =
        "{http://docs.oasis-open.org/ns/opencsa/sca/200903}";
public static final String CONFIDENTIALITY =
        SCA_PREFIX + "confidentiality";
```

```
693        public static final String CONFIDENTIALITY_MESSAGE =
694                CONFIDENTIALITY + ".message";
695
```

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

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

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

703   @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})

704

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

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

```
707        package com.foo;
708
709        import static org.oasisopen.sca.annotation.Confidentiality.*;
710        import static org.oasisopen.sca.annotation.Reference;
711        import static org.oasisopen.sca.annotation.Requires;
712
713        public class Foo {
714           @Requires(CONFIDENTIALITY)
715           @Reference
716           public void setBar(Bar bar) {
717              …
718           }
719        }
720
```

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

```
724        package com.foo;
725
726        import static org.oasisopen.sca.Constants.*;
727        import static org.oasisopen.sca.annotation.Reference;
728        import static org.oasisopen.sca.annotation.Requires;
729
730        public class Foo {
731           @Requires(SCA_PREFIX+"confidentiality")
732           @Reference
733           public void setBar(Bar bar) {
734              …
735           }
736        }
737
```

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

739   '@Requires("' QualifiedIntent '"' (',"' QualifiedIntent '"')* ')'

740   where

741   QualifiedIntent ::= QName('.' Qualifier)*
742   Qualifier ::= NCName

743

744   See section @Requires for the formal definition of the @Requires annotation.

8.2 Specific Intent Annotations

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

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

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

757   `@Integrity`

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

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

760   This annotation attaches the pair of qualified intents: "authentication.message" and
761   "authentication.transport" (the sca: namespace is assumed in this both of these cases –
762   "http://docs.oasis-open.org/ns/opencsa/sca/200903").

763   The general form of specific intent annotations is:

764   '@' Intent ('(' qualifiers ')')?

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

766   Intent   ::= NCName
767   qualifiers   ::= '"' qualifier '"' (',"' qualifier '"')*
768   qualifier   ::= NCName ('.' qualifier)?
769

8.2.1 How to Create Specific Intent Annotations

771   <mark>SCA identifies annotations that correspond to intents by providing an @Intent annotation which</mark>
772   <mark>MUST be used in the definition of a specific intent annotation.</mark> [JCA70001]

773   The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
774   String form of the QName of the intent. As part of the intent definition, it is good practice
775   (although not required) to also create String constants for the Namespace, for the Intent and for
776   Qualified versions of the Intent (if defined). These String constants are then available for use with
777   the @Requires annotation and it is also possible to use one or more of them as parameters to the
778   specific intent annotation.

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

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

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

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

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

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

| 790 | implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport" |
| 791 | are set for the element to which the @Confidentiality annotation is attached. |

792 See section @Qualifier for the formal definition of the @Qualifier annotation.

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

8.3 Application of Intent Annotations

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

797 • Java class

798 • Java interface

799 • Method

800 • Field

801 • Constructor parameter

802 Intent annotations MUST NOT be applied to the following:

803 • A method of a service implementation class, except for a setter method that is either
804 annotated with @Reference or introspected as an SCA reference according to the rules in
805 the appropriate Component Implementation specification

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

809 • A service implementation class constructor parameter that is not annotated with
810 @Reference

811 [JCA70002]

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

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

819 An example of multiple policy annotations being used together follows:

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

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

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

8.3.1 Intent Annotation Examples

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

832 Example 8.1 shows how intents on references are merged. In this example, the intents for myRef
833 are "authentication" and "confidentiality.message".

```
834     @Authentication
```

```
835          @Requires(CONFIDENTIALITY)
836          @Confidentiality("message")
837          @Reference
838          protected MyService myRef;
```
839   Example 8.1. Merging intents on references.

840   Example 8.2 shows that mutually exclusive intents cannot be applied to the same Java element.
841   In this example, the Java code is in error because of contradictory mutually exclusive intents
842   "managedTransaction" and "noManagedTransaction".

```
843          @Requires({SCA_PREFIX+"managedTransaction",
844                     SCA_PREFIX+"noManagedTransaction"})
845          @Reference
846          protected MyService myRef;
```
847   Example 8.2. Mutually exclusive intents.

848   Example 8.3 shows that intents can be applied to Java service interfaces and their methods. In
849   this example, the effective intents for MyService.mymethod() are "authentication" and
850   "confidentiality".

```
851          @Authentication
852          public interface MyService {
853              @Confidentiality
854              public void mymethod();
855          }
856          @Service(MyService.class)
857          public class MyServiceImpl {
858              public void mymethod() {...}
859          }
```
860   Example 8.3. Intents on Java interfaces, interface methods, and Java classes.

861   Example 8.4 shows that intents can be applied to Java service implementation classes. In this
862   example, the effective intents for MyService.mymethod() are "authentication", "confidentiality",
863   and "managedTransaction".

```
864          @Authentication
865          public interface MyService {
866              @Confidentiality
867              public void mymethod();
868          }
869          @Service(MyService.class)
870          @Requires(SCA_PREFIX+"managedTransaction")
871          public class MyServiceImpl {
872              public void mymethod() {...}
873          }
```
874   Example 8.4. Intents on Java service implementation classes.

875   Example 8.5 shows that intents can be applied to Java reference interfaces and their methods,
876   and also to Java references. In this example, the effective intents for the method mymethod() of
877   the reference myRef are "authentication", "integrity", and "confidentiality".

```
878          @Authentication
879          public interface MyRefInt {
880              @Integrity
881              public void mymethod();
882          }
883          @Service(MyService.class)
884          public class MyServiceImpl {
885              @Confidentiality
886              @Reference
887              protected MyRefInt myRef;
888          }
```

889　Example 8.5. Intents on Java references and their interfaces and methods.

890　Example 8.6 shows that intents cannot be applied to methods of Java implementation classes. In
891　this example, the Java code is in error because of the @Authentication intent annotation on the
892　implementation method `MyServiceImpl.mymethod()`.

```
893        public interface MyService {
894            public void mymethod();
895        }
896        @Service(MyService.class)
897        public class MyServiceImpl {
898            @Authentication
899            public void mymethod() {...}
900        }
```

901　Example 8.6. Intent on implementation method.

902　Example 8.7 shows one effect of applying the SCA Policy Framework rules for merging intents
903　within a structural hierarchy to Java service interfaces and their methods. In this example a
904　qualified intent overrides an unqualified intent, so the effective intent for
905　`MyService.mymethod()` is "confidentiality.message".

```
906        @Confidentiality("message")
907        public interface MyService {
908            @Confidentiality
909            public void mymethod();
910        }
```

911　Example 8.7. Merging qualified and unqualified intents on Java interfaces and methods.

912　Example 8.8 shows another effect of applying the SCA Policy Framework rules for merging intents
913　within a structural hierarchy to Java service interfaces and their methods. In this example a
914　lower-level intent causes a mutually exclusive higher-level intent to be ignored, so the effective
915　intent for `mymethod1()` is "managedTransaction" and the effective intent for `mymethod2()` is
916　"noManagedTransaction".

```
917        @Requires(SCA_PREFIX+"managedTransaction")
918        public interface MyService {
919            public void mymethod1();
920            @Requires(SCA_PREFIX+"noManagedTransaction")
921            public void mymethod2();
922        }
```

923　Example 8.8. Merging mutually exclusive intents on Java interfaces and methods.

924　8.3.2 Inheritance and Annotation

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

```
927        package services.hello;
928        import org.oasisopen.sca.annotation.Authentication;
929        import org.oasisopen.sca.annotation.Integrity;
930
931        @Integrity("transport")
932        @Authentication
933        public class HelloService {
934            @Integrity
935            @Authentication("message")
936            public String hello(String message) {...}
937
938            @Integrity
939            @Authentication("transport")
940            public String helloThere() {...}
941        }
```

```
942
943        package services.hello;
944        import org.oasisopen.sca.annotation.Authentication;
945        import org.oasisopen.sca.annotation.Confidentiality;
946
947        @Confidentiality("message")
948        public class HelloChildService extends HelloService {
949              @Confidentiality("transport")
950              public String hello(String message) {...}
951              @Authentication
952              String helloWorld() {...}
953        }
```
954   Example 8.9. Usage example of annotated policy and inheritance.

955

956   The effective intent annotation on the **helloWorld** method of **HelloChildService** is
957   @Authentication and @Confidentiality("message").

958   The effective intent annotation on the **hello** method of **HelloChildService** is
959   @Confidentiality("transport"),

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

962   The effective intent annotation on the **hello** method of **HelloService** is @Integrity and
963   @Authentication("message")

964

965   Table 8.1 below shows the equivalent declarative security interaction policy of the methods of the
966   HelloService and HelloChildService implementations corresponding to the Java classes shown in
967   Example 8.9.

968

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

969
970   Table 8.1. Declarative intents equivalent to annotated intents in Example 8.9.

8.4 Relationship of Declarative and Annotated Intents

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

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

8.5 Policy Set Annotations

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

983

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

987   '@PolicySets({' policySetQName (',' policySetQName)* '})'

988

989 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

990 An example of the @PolicySets annotation:

991

992
```
    @Reference(name="helloService", required=true)
    @PolicySets({ MY_NS + "WS_Encryption_Policy",
                  MY_NS + "WS_Authentication_Policy" })
    public setHelloService(HelloService service) {
          . . .
    }
```
993
994
995
996
997
998

999 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
1000 using the namespace defined for the constant MY_NS.

1001 PolicySets need to satisfy intents expressed for the implementation when both are present,
1002 according to the rules defined in the Policy Framework specification [POLICY].

1003 The SCA Policy Set annotation can be applied to the following Java elements:

1004 • Java class

1005 • Java interface

1006 • Method

1007 • Field

1008 • Constructor parameter

1009 The @PolicySets annotation MUST NOT be applied to the following:

1010 • A method of a service implementation class, except for a setter method that is either
1011   annotated with @Reference or introspected as an SCA reference according to the rules in
1012   the appropriate Component Implementation specification

1013 • A service implementation class field that is not either annotated with @Reference or
1014   introspected as an SCA reference according to the rules in the appropriate Component
1015   Implementation specification

1016 • A service implementation class constructor parameter that is not annotated with
1017   @Reference

1018 [JCA70005]

1019 The @PolicySets annotation can be applied to classes, interfaces, and interface methods. Applying
1020 a @PolicySets annotation to a field, setter method, or constructor parameter allows policy sets to
1021 be defined at references. The @PolicySets annotation can also be applied to reference interfaces
1022 and their methods.

1023 If the @PolicySets annotation is specified on both an interface method and the method's declaring
1024 interface, the SCA runtime MUST compute the effective policy sets for the method by merging the
1025 policy sets from the method with the policy sets from the interface. [JCA70006] This merging
1026 process does not remove or change any policy sets that are applied to the interface.

8.6 Security Policy Annotations

1028 This section introduces annotations for SCA's security intents, as defined in the SCA Policy
1029 Framework specification [POLICY].

1030 ### 8.6.1 Security Interaction Policy

1031 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply
1032 to the operation of services and references of an implementation:

1033 - @Integrity

1034 - @Confidentiality

1035 - @Authentication

1036 All three of these intents have the same pair of Qualifiers:

1037 - message

1038 - transport

1039 The formal definitions of the @Authentication, @Confidentality and @Integrity annotations are
1040 found in the sections @Authentication, @Confidentiality and @Integrity.

1041 The following example shows an example of applying an intent to the setter method used to inject
1042 a reference.   Accessing the hello operation of the referenced HelloService requires both
1043 "integrity.message" and "authentication.message" intents to be honored.

1044

```
1045    package services.hello;
1046    // Interface for HelloService
1047    public interface HelloService {
1048          String hello(String helloMsg);
1049    }
1050
1051    package services.client;
1052    // Interface for ClientService
1053    public interface ClientService {
1054        public void clientMethod();
1055    }
1056
1057    // Implementation class for ClientService
1058    package services.client;
1059
1060    import services.hello.HelloService;
1061    import org.oasisopen.sca.annotation.*;
1062
1063    @Service(ClientService.class)
1064    public class ClientServiceImpl implements ClientService {
1065
1066          private HelloService helloService;
1067
1068          @Reference(name="helloService", required=true)
1069          @Integrity("message")
1070          @Authentication("message")
1071          public void setHelloService(HelloService service) {
1072                helloService = service;
1073          }
1074
1075          public void clientMethod() {
1076                String result = helloService.hello("Hello World!");
```

```
1077                          …
1078                  }
1079           }
1080
```

1081   Example 8.10. Usage of annotated intents on a reference.

9 Java API

1083 This section provides a reference for the Java API offered by SCA.

9.1 Component Context

1085 The following Java code defines the **ComponentContext** interface:

1086

```
1087    package org.oasisopen.sca;
1088    import java.util.Collection;
1089    public interface ComponentContext {

1091        String getURI();

1093        <B> B getService(Class<B> businessInterface, String referenceName);

1095        <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
1096                                                    String referenceName);
1097        <B> Collection<B> getServices(Class<B> businessInterface,
1098            String referenceName);

1100        <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
1101            businessInterface, String referenceName);

1103        <B> ServiceReference<B> createSelfReference(Class<B>
1104            businessInterface);

1106        <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
1107                                                    String serviceName);

1109        <B> B getProperty(Class<B> type, String propertyName);

1111        RequestContext getRequestContext();

1113        <B> ServiceReference<B> cast(B target) throws IllegalArgumentException;

1115    }
```

1116

1117 • **getURI()** - returns the absolute URI of the component within the SCA domain

1118 • **getService(Class<B> businessInterface, String referenceName)** – Returns a proxy for
1119 the reference defined by the current component. The getService() method takes as its
1120 input arguments the Java type used to represent the target service on the client and the
1121 name of the service reference. It returns an object providing access to the service. The
1122 returned object implements the Java interface the service is typed with. The
1123 ComponentContext.getService method MUST throw an IllegalArgumentException if the
1124 reference identified by the referenceName parameter has multiplicity of 0..n or
1125 1..n.[JCA80001]

1126 • **getServiceReference(Class<B> businessInterface, String referenceName)** – Returns a
1127 ServiceReference defined by the current component. This method MUST throw an
1128 IllegalArgumentException if the reference has multiplicity greater than one.

1129 • **getServices(Class<B> businessInterface, String referenceName)** – Returns a list of
1130 typed service proxies for a business interface type and a reference name.

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

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

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

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

- **getRequestContext()** - Returns the context for the current SCA service request, or null if there is no current request or if the context is unavailable. The ComponentContext.getRequestContext method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases. [JCA80002]

- **cast**(**B target**) - Casts a type-safe reference to a ServiceReference

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

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

```
private ComponentContext componentContext;

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

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

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

9.2 Request Context

The following shows the **RequestContext** interface:

```
package org.oasisopen.sca;

import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();

    String getServiceName();
    <CB> ServiceReference<CB> getCallbackReference();
    <CB> CB getCallback();
```

```
1182            <B> ServiceReference<B> getServiceReference();
1183
1184        }
1185
```

1186   The RequestContext interface has the following methods:

- 1187   **_getSecuritySubject()_** – Returns the JAAS Subject of the current request (see the JAAS
- 1188   Reference Guide [JAAS] for details of JAAS)

- 1189   **_getServiceName()_** – Returns the name of the service on the Java implementation the
- 1190   request came in on

- 1191   **_getCallbackReference()_** – Returns a service reference to the callback as specified by the
- 1192   caller. This method returns null when called for a service request whose interface is not
- 1193   bidirectional or when called for a callback request.

- 1194   **_getCallback()_** – Returns a proxy for the callback as specified by the caller. Similar to the
- 1195   getCallbackReference() method, this method returns null when called for a service request
- 1196   whose interface is not bidirectional or when called for a callback request.

- 1197   **_getServiceReference()_** – <mark>When invoked during the execution of a service operation, the
- 1198   getServiceReference method MUST return a ServiceReference that represents the service
- 1199   that was invoked. When invoked during the execution of a callback operation, the
- 1200   getServiceReference method MUST return a ServiceReference that represents the callback
- 1201   that was invoked.</mark> [JCA80003]

1202 9.3 ServiceReference

1203   ServiceReferences can be injected using the @Reference annotation on a field, a setter method, or
1204   constructor parameter taking the type ServiceReference. The detailed description of the usage of
1205   these methods is described in the section on Asynchronous Programming in this document.

1206   The following Java code defines the **_ServiceReference_** interface:

```
1207    package org.oasisopen.sca;
1208
1209    public interface ServiceReference<B> extends java.io.Serializable {
1210
1211        B getService();
1212        Class<B> getBusinessInterface();
1213    }
1214
```

1215   The ServiceReference interface has the following methods:

- 1216   **getService()** - Returns a type-safe reference to the target of this reference. The instance
- 1217   returned is guaranteed to implement the business interface for this reference. The value
- 1218   returned is a proxy to the target that implements the business interface associated with this
- 1219   reference.

- 1220   **getBusinessInterface()** – Returns the Java class for the business interface associated with
- 1221   this reference.

1222 9.4 ServiceRuntimeException

1223   The following snippet shows the **_ServiceRuntimeException_**.

1224

```
1225    package org.oasisopen.sca;
1226
1227    public class ServiceRuntimeException extends RuntimeException {
1228        …
```

1229   `}`

1230

1231   This exception signals problems in the management of SCA component execution.

1232 9.5 ServiceUnavailableException

1233   The following snippet shows the ***ServiceUnavailableException***.

1234
```
1235    package org.oasisopen.sca;
1236
1237    public class ServiceUnavailableException extends ServiceRuntimeException {
1238        …
1239    }
```
1240

1241   This exception signals problems in the interaction with remote services. These are exceptions
1242   that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException
1243   that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since
1244   it most likely requires human intervention

1245 9.6 InvalidServiceException

1246   The following snippet shows the ***InvalidServiceException***.

1247
```
1248    package org.oasisopen.sca;
1249
1250    public class InvalidServiceException extends ServiceRuntimeException {
1251        …
1252    }
```
1253

1254   This exception signals that the ServiceReference is no longer valid. This can happen when the
1255   target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1256   be resolved by retrying the operation and will most likely require human intervention.

1257 9.7 Constants

1258   The SCA ***Constants*** interface defines a number of constant values that are used in the SCA Java
1259   APIs and Annotations. The following snippet shows the Constants interface:

```
1260    package org.oasisopen.sca;
1261
1262    public interface Constants {
1263        String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200903";
1264        String SCA_PREFIX = "{"+SCA_NS+"}";
1265    }
```
1266

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

The following Java code 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 java.lang.annotation.Retention;
import java.lang.annotation.Target;

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

    boolean value() default true;
}
```


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 following 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 MAY be placed on an individual method of a remotable service implementation, on a service implementation class, or on an individual reference for a remotable service. When applied to a reference, it MAY appear anywhere that the @Remotable annotation MAY appear. It MUST NOT appear anywhere else. [JCA90052]

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

1. If the method has an @AllowsPassByReference annotation, the method is marked "allows pass by reference" if and only if the value of the method's annotation is true.

2. Otheriwse, if the class has an @AllowsPassByReference annotation, the method is marked "allows pass by reference" if and only if the value of the class's annotation is true.

1315   3. Otherwise, the method is not marked "allows pass by reference".

1316 The "allows pass by reference" marking of a reference for a remotable service is determined as
1317 follows:

       1. If the reference has an @AllowsPassByReference annotation, the reference is marked
1318
1319   "allows pass by reference" if and only if the value of the reference's annotation is true.

1320   2. Otherwise, if the service implementation class containing the reference has an
1321   @AllowsPassByReference annotation, the reference is marked "allows pass by reference" if
1322   and only if the value of the class's annotation is true.

1323   3. Otherwise, the reference is not marked "allows pass by reference".

1324

1325 The following snippet shows a sample where @AllowsPassByReference is defined for the
1326 implementation of a service method on the Java component implementation class.

1327

```
1328  @AllowsPassByReference
1329  public String hello(String message) {
1330      …
1331  }
```

1332

1333 The following snippet shows a sample where @AllowsPassByReference is defined for a client
1334 reference of a Java component implementation class.

```
1335  @AllowsPassByReference
1336  @Reference
1337  private StockQuoteService stockQuote;
```

1338

10.2 @Authentication
1339

1340 The following Java code defines the **@Authentication** annotation:

1341
```
1342  package org.oasisopen.sca.annotation;

1343
1344  import static java.lang.annotation.ElementType.FIELD;
1345  import static java.lang.annotation.ElementType.METHOD;
1346  import static java.lang.annotation.ElementType.PARAMETER;
1347  import static java.lang.annotation.ElementType.TYPE;
1348  import static java.lang.annotation.RetentionPolicy.RUNTIME;
1349  import static org.oasisopen.sca.Constants.SCA_PREFIX;

1350
1351  import java.lang.annotation.Inherited;
1352  import java.lang.annotation.Retention;
1353  import java.lang.annotation.Target;

1354
1355  @Inherited
1356  @Target({TYPE, FIELD, METHOD, PARAMETER})
1357  @Retention(RUNTIME)
1358  @Intent(Authentication.AUTHENTICATION)
1359  public @interface Authentication {
1360      String AUTHENTICATION = SCA_PREFIX + "authentication";
1361      String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
1362      String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";

1363
1364      /**
1365       * List of authentication qualifiers (such as "message"
```

```
1366          * or "transport").
1367          *
1368          * @return authentication qualifiers
1369          */
1370         @Qualifier
1371         String[] value() default "";
1372     }
```

1373   The **@Authentication** annotation is used to indicate that the invocation requires authentication.
1374   See the section on Application of Intent Annotations for samples and details.

10.3 @Callback

1376   The following Java code defines the **@Callback** annotation:

```
1377
1378     package org.oasisopen.sca.annotation;
1379
1380     import static java.lang.annotation.ElementType.FIELD;
1381     import static java.lang.annotation.ElementType.METHOD;
1382     import static java.lang.annotation.ElementType.TYPE;
1383     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1384     import java.lang.annotation.Retention;
1385     import java.lang.annotation.Target;
1386
1387     @Target({TYPE, METHOD, FIELD})
1388     @Retention(RUNTIME)
1389     public @interface Callback {
1390
1391         Class<?> value() default Void.class;
1392     }
1393
1394
```

1395   The @Callback annotation is used to annotate a service interface or to annotate a Java class (used
1396   to define an interface) with a callback interface by specifying the Java class object of the callback
1397   interface as an attribute.

1398   The @Callback annotation has the following attribute:

1399   •   *value* – the name of a Java class file containing the callback interface

1400

1401   The @Callback annotation can also be used to annotate a method or a field of an SCA
1402   implementation class, in order to have a callback object injected. When used to annotate a
1403   method or a field of an implementation class for injection of a callback object, the@Callback
1404   annotation MUST NOT specify any attributes. [JCA90046]

1405   An example use of the @Callback annotation to declare a callback interface follows:

```
1406     package somepackage;
1407     import org.oasisopen.sca.annotation.Callback;
1408     import org.oasisopen.sca.annotation.Remotable;
1409     @Remotable
1410     @Callback(MyServiceCallback.class)
1411     public interface MyService {
1412
1413         void someMethod(String arg);
1414     }
1415
1416     @Remotable
1417     public interface MyServiceCallback {
```

```
1418
1419        void receiveResult(String result);
1420    }
1421
```

1422   In this example, the implied component type is:

```
1423    <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903" >
1424
1425        <service name="MyService">
1426            <interface.java interface="somepackage.MyService"
1427                    callbackInterface="somepackage.MyServiceCallback"/>
1428        </service>
1429    </componentType>
```

10.4 @ComponentName

1431   The following Java code defines the **@ComponentName** annotation:

1432

```
1433    package org.oasisopen.sca.annotation;
1434
1435    import static java.lang.annotation.ElementType.FIELD;
1436    import static java.lang.annotation.ElementType.METHOD;
1437    import static java.lang.annotation.ElementType.TYPE;
1438    import java.lang.annotation.Retention;
1439    import java.lang.annotation.Target;
1440
1441    @Target({METHOD, FIELD})
1442    @Retention(RUNTIME)
1443    public @interface ComponentName {
1444
1445    }
1446
```

1447   The @ComponentName annotation is used to denote a Java class field or setter method that is
1448   used to inject the component name.

1449   The following snippet shows a component name field definition sample.

1450

```
1451    @ComponentName
1452    private String componentName;
1453
```

1454   The following snippet shows a component name setter method sample.

```
1455
1456    @ComponentName
1457    public void setComponentName(String name) {
1458     //…
1459    }
```

10.5 @Confidentiality

1461   The following Java code defines the **@Confidentiality** annotation:

```
1462
1463    package org.oasisopen.sca.annotation;
1464
1465    import static java.lang.annotation.ElementType.FIELD;
```

```
1466    import static java.lang.annotation.ElementType.METHOD;
1467    import static java.lang.annotation.ElementType.PARAMETER;
1468    import static java.lang.annotation.ElementType.TYPE;
1469    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1470    import static org.oasisopen.sca.Constants.SCA_PREFIX;
1471
1472    import java.lang.annotation.Inherited;
1473    import java.lang.annotation.Retention;
1474    import java.lang.annotation.Target;
1475
1476    @Inherited
1477    @Target({TYPE, FIELD, METHOD, PARAMETER})
1478    @Retention(RUNTIME)
1479    @Intent(Confidentiality.CONFIDENTIALITY)
1480    public @interface Confidentiality {
1481        String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1482        String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1483        String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
1484
1485        /**
1486         * List of confidentiality qualifiers such as "message" or
1487         * "transport".
1488         *
1489         * @return confidentiality qualifiers
1490         */
1491        @Qualifier
1492        String[] value() default "";
1493    }
```
1494   The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.

1495   See the section on Application of Intent Annotations for samples and details.

10.6 @Constructor

1497   The following Java code defines the **@Constructor** annotation:

```
1498
1499    package org.oasisopen.sca.annotation;
1500
1501    import static java.lang.annotation.ElementType.CONSTRUCTOR;
1502    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1503    import java.lang.annotation.Retention;
1504    import java.lang.annotation.Target;
1505
1506    @Target(CONSTRUCTOR)
1507    @Retention(RUNTIME)
1508    public @interface Constructor { }
1509
```

1510   The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1511   Java component implementation. If a constructor of an implementation class is annotated with
1512   @Constructor and the constructor has parameters, each of these parameters MUST have either a
1513   @Property annotation or a @Reference annotation. [JCA90003]

1514   The following snippet shows a sample for the @Constructor annotation.

```
1515
1516    public class HelloServiceImpl implements HelloService {
1517
1518        public HelloServiceImpl(){
```

```
1519        ...
1520        }
1521
1522        @Constructor
1523        public HelloServiceImpl(@Property(name="someProperty")
1524                               String someProperty ){
1525        ...
1526        }
1527
1528         public String hello(String message) {
1529            ...
1530          }
1531     }
```

10.7 @Context

The following Java code defines the **@Context** 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 Context {

}
```

The @Context annotation is used to denote a Java class field or a setter method that is used to inject a composite context for the component. The type of context to be injected is defined by the type of the Java class field or type of the setter method input argument; the type is either **ComponentContext** or **RequestContext**.

The @Context annotation has no attributes.

The following snippet shows a ComponentContext field definition sample.

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

The following snippet shows a RequestContext field definition sample.

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

10.8 @Destroy

The following Java code defines the **@Destroy** annotation:

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

```
1567
1568    import static java.lang.annotation.ElementType.METHOD;
1569    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1570    import java.lang.annotation.Retention;
1571    import java.lang.annotation.Target;
1572
1573    @Target(METHOD)
1574    @Retention(RUNTIME)
1575    public @interface Destroy {
1576
1577    }
1578
```

1579   The @Destroy annotation is used to denote a single Java class method that will be called when the
1580   scope defined for the implementation class ends. A method annotated with @Destroy MAY have
1581   any access modifier and MUST have a void return type and no arguments. [JCA90004]

1582   If there is a method annotated with @Destroy that matches the criteria for the annotation, the
1583   SCA runtime MUST call the annotated method when the scope defined for the implementation
1584   class ends. [JCA90005]

1585   The following snippet shows a sample for a destroy method definition.

1586

```
1587    @Destroy
1588    public void myDestroyMethod() {
1589        …
1590    }
```

10.9 @EagerInit

1592   The following Java code defines the **@EagerInit** annotation:

1593

```
1594    package org.oasisopen.sca.annotation;
1595
1596    import static java.lang.annotation.ElementType.TYPE;
1597    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1598    import java.lang.annotation.Retention;
1599    import java.lang.annotation.Target;
1600
1601    @Target(TYPE)
1602    @Retention(RUNTIME)
1603    public @interface EagerInit {
1604
1605    }
1606
```

1607   The **@EagerInit** annotation is used to mark the Java class of a COMPOSITE scoped
1608   implementation for eager initialization. When marked for eager initialization with an @EagerInit
1609   annotation, the composite scoped instance MUST be created when its containing component is
1610   started. [JCA90007]

10.10 @Init

1612   The following Java code defines the **@Init** annotation:

1613

```
1614    package org.oasisopen.sca.annotation;
1615
1616    import static java.lang.annotation.ElementType.METHOD;
```

```
1617    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1618    import java.lang.annotation.Retention;
1619    import java.lang.annotation.Target;
1620
1621    @Target(METHOD)
1622    @Retention(RUNTIME)
1623    public @interface Init {
1624
1625
1626    }
1627
```

1628   The @Init annotation is used to denote a single Java class method that is called when the scope
1629   defined for the implementation class starts. A method marked with the @Init annotation MAY have
1630   any access modifier and MUST have a void return type and no arguments. [JCA90008]

1631   If there is a method annotated with @Init that matches the criteria for the annotation, the SCA
1632   runtime MUST call the annotated method after all property and reference injection is complete.
1633   [JCA90009]

1634   The following snippet shows an example of an init method definition.

1635

```
1636    @Init
1637    public void myInitMethod() {
1638        …
1639    }
```

1640 10.11 @Integrity

1641   The following Java code defines the **@Integrity** annotation:

```
1642
1643    package org.oasisopen.sca.annotation;
1644
1645    import static java.lang.annotation.ElementType.FIELD;
1646    import static java.lang.annotation.ElementType.METHOD;
1647    import static java.lang.annotation.ElementType.PARAMETER;
1648    import static java.lang.annotation.ElementType.TYPE;
1649    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1650    import static org.oasisopen.sca.Constants.SCA_PREFIX;
1651
1652    import java.lang.annotation.Inherited;
1653    import java.lang.annotation.Retention;
1654    import java.lang.annotation.Target;
1655
1656    @Inherited
1657    @Target({TYPE, FIELD, METHOD, PARAMETER})
1658    @Retention(RUNTIME)
1659    @Intent(Integrity.INTEGRITY)
1660    public @interface Integrity {
1661        String INTEGRITY = SCA_PREFIX + "integrity";
1662        String INTEGRITY_MESSAGE = INTEGRITY + ".message";
1663        String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
1664
1665        /**
1666         * List of integrity qualifiers (such as "message" or "transport").
1667         *
1668         * @return integrity qualifiers
1669         */
```

```
1670          @Qualifier
1671          String[] value() default "";
1672      }
1673
```

1674   The **@Integrity** annotation is used to indicate that the invocation requires integrity (i.e. no
1675   tampering of the messages between client and service).

1676   See the section on Application of Intent Annotations for samples and details.

10.12 @Intent

1678   The following Java code defines the **@Intent** annotation:

```
1679
1680      package org.oasisopen.sca.annotation;
1681
1682      import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1683      import static java.lang.annotation.RetentionPolicy.RUNTIME;
1684      import java.lang.annotation.Retention;
1685      import java.lang.annotation.Target;
1686
1687      @Target({ANNOTATION_TYPE})
1688      @Retention(RUNTIME)
1689      public @interface Intent {
1690          /**
1691           * The qualified name of the intent, in the form defined by
1692           * {@link javax.xml.namespace.QName#toString}.
1693           * @return the qualified name of the intent
1694           */
1695          String value() default "";
1696
1697          /**
1698           * The XML namespace for the intent.
1699           * @return the XML namespace for the intent
1700           */
1701          String targetNamespace() default "";
1702
1703          /**
1704           * The name of the intent within its namespace.
1705           * @return name of the intent within its namespace
1706           */
1707          String localPart() default "";
1708      }
1709
```

1710   The @Intent annotation is used for the creation of new annotations for specific intents. It is not
1711   expected that the @Intent annotation will be used in application code.

1712   See the section "How to Create Specific Intent Annotations" for details and samples of how to
1713   define new intent annotations.

10.13 @OneWay

1715   The following Java code defines the **@OneWay** annotation:

```
1716
1717      package org.oasisopen.sca.annotation;
1718
1719      import static java.lang.annotation.ElementType.METHOD;
1720      import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
1721    import java.lang.annotation.Retention;
1722    import java.lang.annotation.Target;
1723
1724    @Target(METHOD)
1725    @Retention(RUNTIME)
1726    public @interface OneWay {
1727
1728
1729    }
1730
```

1731   The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1732   will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1733   Programming.

1734   The @OneWay annotation has no attributes.

1735   The following snippet shows the use of the @OneWay annotation on an interface.

```
1736    package services.hello;
1737
1738    import org.oasisopen.sca.annotation.OneWay;
1739
1740    public interface HelloService {
1741        @OneWay
1742        void hello(String name);
1743    }
```

10.14 @PolicySets

1745   The following Java code defines the **@PolicySets** annotation:

```
1746
1747    package org.oasisopen.sca.annotation;
1748
1749    import static java.lang.annotation.ElementType.FIELD;
1750    import static java.lang.annotation.ElementType.METHOD;
1751    import static java.lang.annotation.ElementType.PARAMETER;
1752    import static java.lang.annotation.ElementType.TYPE;
1753    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1754
1755    import java.lang.annotation.Retention;
1756    import java.lang.annotation.Target;
1757
1758    @Target({TYPE, FIELD, METHOD, PARAMETER})
1759    @Retention(RUNTIME)
1760    public @interface PolicySets {
1761        /**
1762         * Returns the policy sets to be applied.
1763         *
1764         * @return the policy sets to be applied
1765         */
1766        String[] value() default "";
1767    }
1768
```

1769   The **@PolicySets** annotation is used to attach one or more SCA Policy Sets to a Java
1770   implementation class or to one of its subelements.

1771   See the section "Policy Set Annotations" for details and samples.

10.15 @Property

The following Java code defines the **@Property** 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;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

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

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

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

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

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

The @Property annotation has the following attributes:

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

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

The following snippet shows a property field definition sample.


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


The following snippet shows a property setter sample


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

    ....

}
```

1820　For a @Property annotation, if the type of the Java class field or the type of the input parameter of
1821　the setter method or constructor is defined as an array or as any type that extends or implements
1822　java.util.Collection, then the SCA runtime MUST introspect the component type of the
1823　implementation with a <property/> element with a @many attribute set to true, otherwise
1824　@many MUST be set to false.[JCA90047]

1825　The following snippet shows the definition of a configuration property using the @Property
1826　annotation for a collection.

```
1827    ...
1828    private List<String> helloConfigurationProperty;
1829
1830    @Property(required=true)
1831    public void setHelloConfigurationProperty(List<String> property) {
1832            helloConfigurationProperty = property;
1833    }
1834    ...
```

10.16 @Qualifier

1836　The following Java code defines the **@Qualifier** annotation:

```
1838    package org.oasisopen.sca.annotation;

1840    import static java.lang.annotation.ElementType.METHOD;
1841    import static java.lang.annotation.RetentionPolicy.RUNTIME;

1843    import java.lang.annotation.Retention;
1844    import java.lang.annotation.Target;

1846    @Target(METHOD)
1847    @Retention(RUNTIME)
1848    public @interface Qualifier {
1849    }
```

1851　The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,
1852　defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the
1853　intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the
1854　intent has qualifiers. [JCA90015]

1855　See the section "How to Create Specific Intent Annotations" for details and samples of how to
1856　define new intent annotations.

10.17 @Reference

1858　The following Java code defines the **@Reference** annotation:

```
1860    package org.oasisopen.sca.annotation;

1862    import static java.lang.annotation.ElementType.FIELD;
1863    import static java.lang.annotation.ElementType.METHOD;
1864    import static java.lang.annotation.ElementType.PARAMETER;
1865    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1866    import java.lang.annotation.Retention;
1867    import java.lang.annotation.Target;
1868    @Target({METHOD, FIELD, PARAMETER})
1869    @Retention(RUNTIME)
1870    public @interface Reference {
```

```
1871
1872        String name() default "";
1873        boolean required() default true;
1874    }
1875
```

1876   The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1877   constructor parameter that is used to inject a service that resolves the reference. The interface of
1878   the service injected is defined by the type of the Java class field or the type of the input parameter
1879   of the setter method or constructor.

1880   <mark>The @Reference annotation MUST NOT be used on a class field that is declared as final.</mark>
1881   [JCA90016]

1882   Where there is both a setter method and a field for a reference, the setter method is used.

1883   The @Reference annotation has the following attributes:

- 1884 *name : String (optional)* – the name of the reference. For a field annotation, the default is
  1885 the name of the field of the Java class. For a setter method annotation, the default is the
  1886 JavaBeans property name corresponding to the setter method name. <mark>For a @Reference
  1887 annotation applied to a constructor parameter, there is no default for the name attribute
  1888 and the name attribute MUST be present.</mark> [JCA90018]

- 1889 *required (optional)* – a boolean value which specifies whether injection of the service
  1890 reference is required or not, where true means injection is required and false means
  1891 injection is not required. Defaults to true. <mark>For a @Reference annotation applied to a
  1892 constructor parameter, the required attribute MUST have the value true.</mark> [JCA90019]

1893

1894   The following snippet shows a reference field definition sample.

1895

```
1896    @Reference(name="stockQuote", required=true)
1897    protected StockQuoteService stockQuote;
```

1898

1899   The following snippet shows a reference setter sample

1900

```
1901    @Reference(name="stockQuote", required=true)
1902    public void setStockQuote( StockQuoteService theSQService ) {
1903        ...
1904    }
```

1905

1906   The following fragment from a component implementation shows a sample of a service reference
1907   using the @Reference annotation. The name of the reference is "helloService" and its type is
1908   HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1909   helloService reference.

1910

```
1911    package services.hello;
1912
1913    private HelloService helloService;
1914
1915    @Reference(name="helloService", required=true)
1916    public setHelloService(HelloService service) {
1917        helloService = service;
1918    }
```

```
1919
1920      public void clientMethod() {
1921              String result = helloService.hello("Hello World!");
1922              …
1923      }
1924
```

1925   The presence of a @Reference annotation is reflected in the componentType information that the
1926   runtime generates through reflection on the implementation class. The following snippet shows
1927   the component type for the above component implementation fragment.

```
1928
1929      <?xml version="1.0" encoding="ASCII"?>
1930      <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">
1931
1932          <!-- Any services offered by the component would be listed here -->
1933          <reference name="helloService" multiplicity="1..1">
1934              <interface.java interface="services.hello.HelloService"/>
1935          </reference>
1936
1937      </componentType>
1938
```

1939   If the type of a reference is not an array or any type that extends or implements
1940   java.util.Collection, then the SCA runtime MUST introspect the component type of the
1941   implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference
1942   annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation
1943   required attribute is true. [JCA90020]

1944   If the type of a reference is defined as an array or as any type that extends or implements
1945   java.util.Collection, then the SCA runtime MUST introspect the component type of the
1946   implementation with a <reference/> element with @multiplicity=0..n if the @Reference
1947   annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation
1948   required attribute is true. [JCA90021]

1949   The following fragment from a component implementation shows a sample of a service reference
1950   definition using the @Reference annotation on a java.util.List. The name of the reference is
1951   "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1952   services referenced by the helloServices reference. In this case, at least one HelloService needs
1953   to be present, so *required* is true.

```
1954
1955      @Reference(name="helloServices", required=true)
1956      protected List<HelloService> helloServices;
1957
1958      public void clientMethod() {
1959
1960              …
1961              for (int index = 0; index < helloServices.size(); index++) {
1962                  HelloService helloService =
1963                          (HelloService)helloServices.get(index);
1964                  String result = helloService.hello("Hello World!");
1965              }
1966              …
1967      }
1968
```

1969   The following snippet shows the XML representation of the component type reflected from for the
1970   former component implementation fragment. There is no need to author this component type in
1971   this case since it can be reflected from the Java class.

1972

```
1973    <?xml version="1.0" encoding="ASCII"?>
1974    <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903">
1975
1976        <!-- Any services offered by the component would be listed here -->
1977        <reference name="helloServices" multiplicity="1..n">
1978            <interface.java interface="services.hello.HelloService"/>
1979        </reference>
1980
1981    </componentType>
```

1983 An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by
1984 the SCA runtime as null [JCA90022] An unwired reference with a multiplicity of 0..n MUST be
1985 presented to the implementation code by the SCA runtime as an empty array or empty collection
1986 [JCA90023]

10.17.1 Reinjection

1988 References MAY be reinjected by an SCA runtime after the initial creation of a component if the
1989 reference target changes due to a change in wiring that has occurred since the component was
1990 initialized.   [JCA90024]

1991 In order for reinjection to occur, the following MUST be true:

1992   1.   The component MUST NOT be STATELESS scoped.

1993   2.   The reference MUST use either field-based injection or setter injection. References that
1994   are injected through constructor injection MUST NOT be changed.

1995 [JCA90025]

1996 Setter injection allows for code in the setter method to perform processing in reaction to a change.

1997 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1998 work as if the reference target was not changed. [JCA90026]

1999 If an operation is called on a reference where the target of that reference has been undeployed,
2000 the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called
2001 on a reference where the target of the reference has become unavailable for some reason, the
2002 SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of
2003 the reference is changed, the reference MUST either continue to work or throw an
2004 InvalidServiceException when it is invoked. [JCA90029] If it doesn't work, the exception thrown
2005 will depend on the runtime and the cause of the failure.

2006 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
2007 corresponds to the reference that is passed as a parameter to cast(). If the reference is
2008 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
2009 to work as if the reference target was not changed. [JCA90030] If the target of a ServiceReference
2010 has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an
2011 operation is invoked on the ServiceReference. [JCA90031] If the target of a ServiceReference has
2012 become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an
2013 operation is invoked on the ServiceReference. [JCA90032] If the target service of a
2014 ServiceReference is changed, the reference MUST either continue to work or throw an
2015 InvalidServiceException when it is invoked. [JCA90033] If it doesn't work, the exception thrown
2016 will depend on the runtime and the cause of the failure.

2017 A reference or ServiceReference accessed through the component context by calling getService()
2018 or getServiceReference() MUST correspond to the current configuration of the domain. This applies
2019 whether or not reinjection has taken place. [JCA90034] If the target of a reference or
2020 ServiceReference accessed through the component context by calling getService() or
2021 getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a
2022 reference to the undeployed or unavailable service, and attempts to call business methods
2023 SHOULD throw an InvalidServiceException or a ServiceUnavailableException. [JCA90035] If the
2024 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. [JCA90036]

The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This means that in the cases where reference reinjection is not allowed, the array or Collection for a reference 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. [JCA90037] 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. [JCA90038] A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component. [JCA90039]

	Effect on		
Change event	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().

10.18 @Remotable

The following Java code defines the *@Remotable* annotation:

```
2041        package org.oasisopen.sca.annotation;
2042
2043        import static java.lang.annotation.ElementType.TYPE;
2044        import static java.lang.annotation.RetentionPolicy.RUNTIME;
2045        import java.lang.annotation.Retention;
2046        import java.lang.annotation.Target;
2047
2048
2049        @Target(TYPE)
2050        @Retention(RUNTIME)
2051        public @interface Remotable {
2052
2053        }
2054
```

2055 The @Remotable annotation is used to annotate a Java service interface or to annotate a Java
2056 class (used to define an interface) as remotable. A remotable service can be published externally
2057 as a service and MUST be translatable into a WSDL portType. [JCA90040]

2058 The @Remotable annotation has no attributes.

2059 The following snippet shows the Java interface for a remotable service with its @Remotable
2060 annotation.

```
2061        package services.hello;
2062
2063        import org.oasisopen.sca.annotation.*;
2064
2065        @Remotable
2066        public interface HelloService {
2067
2068            String hello(String message);
2069        }
2070
```

2071 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
2072 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

2073 Complex data types exchanged via remotable service interfaces need to be compatible with the
2074 marshalling technology used by the service binding. For example, if the service is going to be
2075 exposed using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types
2076 or they can be Service Data Objects (SDOs) [SDO].

2077 Independent of whether the remotable service is called from outside of the composite that
2078 contains it or from another component in the same composite, the data exchange semantics are
2079 **by-value**.

2080 Implementations of remotable services can modify input data during or after an invocation and
2081 can modify return data after the invocation. If a remotable service is called locally or remotely, the
2082 SCA container is responsible for making sure that no modification of input data or post-invocation
2083 modifications to return data are seen by the caller.

2084 The following snippet shows a remotable Java service interface.

2085

```
2086        package services.hello;
2087
2088        import org.oasisopen.sca.annotation.*;
2089
2090        @Remotable
2091        public interface HelloService {
2092
2093            String hello(String message);
```

```
2094          }
2095
2096      package services.hello;
2097
2098      import org.oasisopen.sca.annotation.*;
2099
2100      @Service(HelloService.class)
2101      public class HelloServiceImpl implements HelloService {
2102
2103          public String hello(String message) {
2104                  ...
2105          }
2106      }
```

2107 10.19 @Requires

2108 The following Java code defines the **@Requires** annotation:

```
2109
2110      package org.oasisopen.sca.annotation;
2111
2112      import static java.lang.annotation.ElementType.FIELD;
2113      import static java.lang.annotation.ElementType.METHOD;
2114      import static java.lang.annotation.ElementType.PARAMETER;
2115      import static java.lang.annotation.ElementType.TYPE;
2116      import static java.lang.annotation.RetentionPolicy.RUNTIME;
2117
2118      import java.lang.annotation.Inherited;
2119      import java.lang.annotation.Retention;
2120      import java.lang.annotation.Target;
2121
2122      @Inherited
2123      @Retention(RUNTIME)
2124      @Target({TYPE, METHOD, FIELD, PARAMETER})
2125      public @interface Requires {
2126          /**
2127           * Returns the attached intents.
2128           *
2129           * @return the attached intents
2130           */
2131          String[] value() default "";
2132      }
2133
```

2134 The **@Requires** annotation supports general purpose intents specified as strings. Users can also
2135 define specific intent annotations using the @Intent annotation.

2136 See the section "General Intent Annotations" for details and samples.

2137 10.20 @Scope

2138 The following Java code defines the **@Scope** annotation:

```
2139      package org.oasisopen.sca.annotation;
2140
2141      import static java.lang.annotation.ElementType.TYPE;
2142      import static java.lang.annotation.RetentionPolicy.RUNTIME;
2143      import java.lang.annotation.Retention;
2144      import java.lang.annotation.Target;
2145
```

```
2146    @Target(TYPE)
2147    @Retention(RUNTIME)
2148    public @interface Scope {
2149
2150        String value() default "STATELESS";
2151    }
```
2152   The @Scope annotation MUST only be used on a service's implementation class. It is an error to
2153   use this annotation on an interface. [JCA90041]

2154   The @Scope annotation has the following attribute:

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

2161   The default value is STATELESS.

2162   The following snippet shows a sample for a COMPOSITE scoped service implementation:

```
2163    package services.hello;
2164
2165    import org.oasisopen.sca.annotation.*;
2166
2167    @Service(HelloService.class)
2168    @Scope("COMPOSITE")
2169    public class HelloServiceImpl implements HelloService {
2170
2171        public String hello(String message) {
2172            ...
2173        }
2174    }
2175
```

2176   10.21 @Service

2177   The following Java code defines the **@Service** annotation:

```
2178    package org.oasisopen.sca.annotation;
2179
2180    import static java.lang.annotation.ElementType.TYPE;
2181    import static java.lang.annotation.RetentionPolicy.RUNTIME;
2182    import java.lang.annotation.Retention;
2183    import java.lang.annotation.Target;
2184
2185    @Target(TYPE)
2186    @Retention(RUNTIME)
2187    public @interface Service {
2188
2189        Class<?>[] interfaces() default { Void.class };
2190        String name() default "";
2191        String[] names() default {};
2192        Class<?> value() default Void.class;
2193    }
2194
```

2195   The @Service annotation is used on a component implementation class to specify the SCA services
2196   offered by the implementation. An implementation class need not be declared as implementing all
2197   of the interfaces implied by the services declared in its @Service annotation, but all methods of all
2198   the declared service interfaces MUST be present. [JCA90042] A class used as the implementation

of a service is not required to have a @Service annotation. If a class has no @Service annotation, then the rules determining which services are offered and what interfaces those services have are determined by the specific implementation type.

The @Service annotation has the following attributes:

- *interfaces (1..1)* – The value is an array of interface or class objects that are exposed as services by this implementation.

- *name (0..1)* - A string which is used as the service name. If the name attribute is specified on the @Service annotation, the value attribute MUST also be specified. [JCA90048]

- *names (0..1)* - Contains an array of Strings which are used as the service names for each of the interfaces declared in the *interfaces* array. If the names attribute is specified for an @Service annotation, the interfaces attribute MUST also be specified. [JCA90049] The number of Strings in the names attributes array of the @Service annotation MUST match the number of elements in the interfaces attribute array. [JCA90050]

- *value* – A shortcut for the case when the class provides only a single service interface - contains a single interface or class object that is exposed as a service by this component implementation.

A @Service annotation MUST only have one of the interfaces attribute or value attribute specified. [JCA90043]

A @Service annotation that specifies a single class object Void.class either explicitly or by default is equivalent to not having the annotation there at all - such a @Service annotation MUST be ignored. [JCA90044] The @Service annotation MUST NOT specify Void.class in conjunction with any other service class or interface. [JCA90051]

The *service names* of the defined services default to the names of the interfaces or class, without the package name. If the names parameter is specified, the service name for each interface in the interfaces attribute array is the String declared in the corresponding position in the names attribute array.

A component implementation MUST NOT have two services with the same Java simple name. [JCA90045] If a Java implementation needs to realize two services with the same Java simple name then this can be achieved through subclassing of the interface.

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

```
package services.hello;

import org.oasisopen.sca.annotation.Service;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {

    public void hello(String name) {
        System.out.println("Hello " + name);
    }
}
```

11 WSDL to Java and Java to WSDL

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

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

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

11.1 JAX-WS Client Asynchronous API for a Synchronous Service

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

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

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

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

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

    b. the same parameter signature as M, and

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

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

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

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

    b. a parameter signature that is M's parameter signature with an additional final parameter of type AsyncHandler<R> where R is the return type of M, and

    c. a return type of Future<?>

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

As an example, an interface can be defined in WSDL as follows:

```
2289   <!-- WSDL extract -->
2290   <message name="getPrice">
2291    <part name="ticker" type="xsd:string"/>
2292   </message>
2293
2294   <message name="getPriceResponse">
2295    <part name="price" type="xsd:float"/>
2296   </message>
2297
2298   <portType name="StockQuote">
2299    <operation name="getPrice">
2300        <input message="tns:getPrice"/>
2301        <output message="tns:getPriceResponse"/>
2302    </operation>
2303   </portType>
```

2304

2305   The JAX-WS asynchronous mapping will produce the following Java interface:

```
2306   // asynchronous mapping
2307   @WebService
2308   public interface StockQuote {
2309    float getPrice(String ticker);
2310    Response<Float> getPriceAsync(String ticker);
2311    Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2312   }
```

2313

2314   For SCA interface definition purposes, this is treated as equivalent to the following:

```
2315   // synchronous mapping
2316   @WebService
2317   public interface StockQuote {
2318    float getPrice(String ticker);
2319   }
```

2320

2321   SCA runtimes MUST support the use of the JAX-WS client asynchronous model. [JCA100009] In
2322   the above example, if the client implementation uses the asynchronous form of the interface, the
2323   two additional getPriceAsync() methods can be used for polling and callbacks as defined by the
2324   JAX-WS specification.

12 Conformance

The XML schema pointed to by the RDDL 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.

For code artifacts related to this specification, the specification text is considered to be authoritative and takes precedence over the code artifacts.

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, an SCA ComponentType Document or an SCA ConstrainingType 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 or SCA ConstrainingType 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_CI].

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.xsd

2363

```xml
2364  <?xml version="1.0" encoding="UTF-8"?>
2365  <!-- Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
2366      OASIS trademark, IPR and other policies apply.  -->
2367  <schema xmlns="http://www.w3.org/2001/XMLSchema"
2368     targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200903"
2369     xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903"
2370     elementFormDefault="qualified">
2371
2372     <include schemaLocation="sca-core-1.1-cd03.xsd"/>
2373
2374     <!-- Java Interface -->
2375     <element name="interface.java" type="sca:JavaInterface"
2376            substitutionGroup="sca:interface"/>
2377     <complexType name="JavaInterface">
2378        <complexContent>
2379           <extension base="sca:Interface">
2380              <sequence>
2381                 <any namespace="##other" processContents="lax" minOccurs="0"
2382                    maxOccurs="unbounded"/>
2383              </sequence>
2384              <attribute name="interface" type="NCName" use="required"/>
2385              <attribute name="callbackInterface" type="NCName"
2386                       use="optional"/>
2387              <anyAttribute namespace="##other" processContents="lax"/>
2388           </extension>
2389        </complexContent>
2390     </complexType>
2391
2392  </schema>
2393
```

B. Conformance Items

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

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of *method overloading*.
[JCA20002]	the SCA runtime MUST 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 MUST 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 MUST 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 MUST 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 MUST call that method when the implementation instance is created.
[JCA20007]	the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT 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 MUST 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 MAY 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 MUST 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 MUST be the fully qualified name of the Java interface class
[JCA30002]	The value of the @callbackInterface attribute MUST 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.

[JCA30008]	A Java implementation class referenced by the @interface or the @callbackInterface attribute of an <interface.java/> element MUST NOT contain the following SCA Java annotations: @Intent, @Qualifier.
[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.
[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.
[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 getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.
[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.
[JCA90002]	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 MAY 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 MAY 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 have the value true.
[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 a 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.
[JCA90043]	A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.
[JCA90044]	A @Service annotation that specifies a single class object Void.class either explicitly or by default is equivalent to not having the annotation there at all - such a @Service annotation MUST be ignored.
[JCA90045]	A component implementation MUST NOT have two services with the same Java simple name.
[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.
[JCA90048]	If the name attribute is specified on the @Service annotation, the value attribute MUST also be specified.
[JCA90049]	If the names attribute is specified for an @Service annotation, the interfaces attribute MUST also be specified.
[JCA90050]	The number of Strings in the names attributes array of the @Service annotation MUST match the number of elements in the interfaces attribute array.
[JCA90051]	The @Service annotation MUST NOT specify Void.class in conjunction with any other service class or interface.
[JCA90052]	The @AllowsPassByReference annotation MAY be placed on an individual method of a remotable service implementation, on a service implementation class, or on an individual reference for a remotable service. When applied to a reference, it MAY appear anywhere that the @Remotable annotation MAY appear. It MUST NOT appear anywhere else.
[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 Java types to XML schema types.

[JCA100005]	SCA runtimes MAY support the SDO 2.1 mapping from Java types to XML schema types.
[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 Java interface MAY contain 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.

2398

Scott Vorthmann   TIBCO Software Inc.
Feng Wang   Primeton Technologies, Inc.
Robin Yang   Primeton Technologies, Inc.

2403

2404

D. Non-Normative Text

2405

E. Revision History

2407   [optional; should not be included in OASIS Standards]

2408

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 Combellack	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

2409