



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 03+Issue1

16 March 2009

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.pdf> (normative)

Previous Version:

Latest Version:

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

Latest Approved Version:

Technical Committee:

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

Chair(s):

Simon Nash,	IBM
Michael Rowley,	BEA Systems
Mark Combella,	Avaya

Editor(s):

Ron Barack,	SAP
David Booz,	IBM
Mark Combella,	Avaya
Mike Edwards,	IBM
Anish Karmarkar,	Oracle
Ashok Malhotra,	Oracle
Peter Peshev,	SAP

Related work:

This specification replaces or supersedes:

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

This specification is related to:

- Service Component Architecture Assembly Model Specification Version 1.1
- Service Component Architecture Policy Framework Specification Version 1.1

Declared XML Namespace(s):

<http://docs.oasis-open.org/ns/opencsa/sca/200712>

Abstract:

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

Specifically, this specification covers:

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

Note that individual programming models can 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, 2008. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction.....	7
1.1	Terminology.....	7
1.2	Normative References.....	7
1.3	Non-Normative References.....	8
2	Implementation Metadata.....	9
2.1	Service Metadata.....	9
2.1.1	@Service.....	9
2.1.2	Java Semantics of a Remotable Service.....	9
2.1.3	Java Semantics of a Local Service.....	9
2.1.4	@Reference.....	10
2.1.5	@Property.....	10
2.2	Implementation Scopes: @Scope, @Init, @Destroy.....	10
2.2.1	Stateless scope.....	10
2.2.2	Composite scope.....	11
3	Interface.....	12
3.1	Java interface element – <interface.java>.....	12
3.2	@Remotable.....	13
3.3	@Callback.....	13
4	Client API.....	14
4.1	Accessing Services from an SCA Component.....	14
4.1.1	Using the Component Context API.....	14
4.2	Accessing Services from non-SCA component implementations.....	14
4.2.1	SCAClient Interface and Related Classes.....	14
5	Error Handling.....	15
6	Asynchronous Programming.....	16
6.1	@OneWay.....	16
6.2	Callbacks.....	16
6.2.1	Using Callbacks.....	16
6.2.2	Callback Instance Management.....	18
6.2.3	Implementing Multiple Bidirectional Interfaces.....	18
6.2.4	Accessing Callbacks.....	19
7	Policy Annotations for Java.....	20
7.1	General Intent Annotations.....	20
7.2	Specific Intent Annotations.....	22
7.2.1	How to Create Specific Intent Annotations.....	22
7.3	Application of Intent Annotations.....	23
7.3.1	Inheritance And Annotation.....	23
7.4	Relationship of Declarative And Annotated Intents.....	25
7.5	Policy Set Annotations.....	25
7.6	Security Policy Annotations.....	26
7.6.1	Security Interaction Policy.....	26
7.6.2	Security Implementation Policy.....	27
8	Java API.....	30

8.1 Component Context.....	30
8.2 Request Context.....	31
8.3 ServiceReference.....	32
8.4 ServiceRuntimeException.....	32
8.5 ServiceUnavailableException.....	33
8.6 InvalidServiceException.....	33
8.7 Constants.....	33
8.8 SCAClient Interface.....	33
8.9 SCAClientFactory Class.....	34
8.10 NoSuchDomainException.....	36
8.11 NoSuchServiceException.....	36
9 Java Annotations.....	37
9.1 @AllowsPassByReference.....	37
9.2 @Authentication.....	37
9.3 @Callback.....	38
9.4 @ComponentName.....	39
9.5 @Confidentiality.....	40
9.6 @Constructor.....	41
9.7 @Context.....	41
9.8 @Destroy.....	42
9.9 @EagerInit.....	42
9.10 @Init.....	43
9.11 @Integrity.....	43
9.12 @Intent.....	44
9.13 @OneWay.....	45
9.14 @PolicySets.....	45
9.15 @Property.....	46
9.16 @Qualifier.....	47
9.17 @Reference.....	48
9.17.1 Reinjection.....	50
9.18 @Remotable.....	52
9.19 @Requires.....	53
9.20 @Scope.....	54
9.21 @Service.....	55
10 WSDL to Java and Java to WSDL.....	57
10.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	57
A. XML Schema: sca-interface-java.xsd.....	59
B. Java Classes and Interfaces.....	60
B.1 SCAClient Classes and Interfaces.....	60
B.1.1 SCAClient Interface.....	60
B.1.2 SCAClientFactory Class.....	61
B.1.3 SCAFactoryFinder class.....	62
B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?.....	67
C. Conformance Items.....	69
D. Acknowledgements.....	75

E. Non-Normative Text	76
F. Revision History.....	77

1 Introduction

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

Specifically, this specification covers:

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

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

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

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

Comment [ME1]: This sentence needs to be removed

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 Specification, <http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-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>, WSDL 2.0: <http://www.w3.org/TR/wsd20/>
- [POLICY] SCA Policy Framework, <http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf>

- 44 **[JSR-250]** Common Annotation for Java Platform specification (JSR-250),
45 <http://www.jcp.org/en/jsr/detail?id=250>
46 **[JAX-WS]** JAX-WS 2.1 Specification (JSR-224),
47 <http://www.jcp.org/en/jsr/detail?id=224>
48 **[JAVABEANS]** JavaBeans 1.01 Specification,
49 <http://java.sun.com/javase/technologies/desktop/javabeans/api/>
50

51 **1.3 Non-Normative References**

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

54 2 Implementation Metadata

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

57 2.1 Service Metadata

58 2.1.1 @Service

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

- 61 • As a Java interface
- 62 • As a Java class
- 63 • As a Java interface generated from a Web Services Description Language [WSDL]
64 (WSDL) portType (Java interfaces generated from a WSDL portType are always
65 **remotable**)

66 2.1.2 Java Semantics of a Remotable Service

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

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

```
72 package services.hello;  
73 @Remotable  
74 public interface HelloService {  
75     String hello(String message);  
76 }
```

77 2.1.3 Java Semantics of a Local Service

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

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

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

```
83 package services.hello;  
84 public interface HelloService {  
85     String hello(String message);  
86 }  
87
```

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

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

94 2.1.4 @Reference

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

98 2.1.5 @Property

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

102 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

109 This document defines two scopes:

- 110 • STATELESS
- 111 • COMPOSITE

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

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

116 **@Init** denotes a method called upon first use of an instance during the lifetime of the scope
117 (except for composite scoped implementation marked to eagerly initialize, see [section Composite](#)
118 [Scope](#)).

119 **@Destroy** specifies a method called when the scope ends.

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

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

```
124     @Init  
125     public void start() {  
126         ...  
127     }  
128  
129     @Destroy  
130     public void stop() {  
131         ...  
132     }  
133  
134
```

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

137 2.2.1 Stateless scope

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

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

146 2.2.2 Composite scope

147 For a composite scope implementation instance, the SCA runtime MUST ensure that all service
148 requests are dispatched to the same implementation instance for the lifetime of the containing
149 composite. [JCA20004] The lifetime of the containing composite is defined as the time it becomes
150 active in the runtime to the time it is deactivated, either normally or abnormally.

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

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

158 3 Interface

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

160 3.1 Java interface element – <interface.java>

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

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

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

169

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

171

172 The `interface.java` element has the following attributes:

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

179

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

181

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

184

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

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

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

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

199

200

201 3.2 @Remotable

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

206 3.3 @Callback

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

211 4 Client API

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

214 4.1 Accessing Services from an SCA Component

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

219 4.1.1 Using the Component Context API

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

223 4.2 Accessing Services from non-SCA component implementations

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

226 4.2.1 SCAClient Interface and Related Classes

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

230 Objects which implement the SCAClient interface are obtained using the SCAClientFactory class.

231 The following is a sample of the code that a client would use:

```
232 import org.oasisopen.sca.client;  
233 import com.foo.HelloService;  
234  
235 public void someMethod() {  
236     String serviceURI = "SomeHelloServiceURI";  
237     URI domainURI = new URI("SomeDomainURI");  
238     ...  
239     SCAClient scaClient = SCAClientFactory.newInstance();  
240     HelloService helloService =  
241         scaClient.getService(HelloService.class,  
242                              serviceURI, domainURI);  
243     String reply = helloService.sayHello("Mark");  
244     ...  
245 }
```

246 For details about the SCAClient interface and its related classes see the section "SCAClient
247 Interface" and the section "SCAClientFactory Class".

251

Deleted: ComponentContext

Deleted: Non-SCA client code can use the ComponentContext API to perform operations against a component in an SCA domain. How client code obtains a reference to a ComponentContext is runtime specific. ¶ The following example demonstrates the use of the component Context API by non-SCA code:

Deleted: ¶

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

252 5 Error Handling

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

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

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

259 6 Asynchronous Programming

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

- 267 • support for non-blocking method calls
- 268 • callbacks

269 Each of these topics is discussed in the following sections.

270 6.1 @OneWay

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

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

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

281 6.2 Callbacks

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

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

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

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

295 6.2.1 Using Callbacks

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

301 The following example shows a scenario in which bidirectional interfaces and callbacks could be
302 used. A client requests a quotation from a supplier. To process the enquiry and return the

303 quotation, some suppliers might need additional information from the client. The client does not
304 know which additional items of information will be needed by different suppliers. This interaction
305 can be modeled as a bidirectional interface with callback requests to obtain the additional
306 information.

```
307 package somepackage;  
308 import org.osoa.sca.annotation.Callback;  
309 import org.osoa.sca.annotation.Remotable;  
310 @Remotable  
311 @Callback(QuotationCallback.class)  
312 public interface Quotation {  
313     double requestQuotation(String productCode, int quantity);  
314 }  
315  
316 @Remotable  
317 public interface QuotationCallback {  
318     String getState();  
319     String getZipCode();  
320     String getCreditRating();  
321 }  
322
```

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

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

```
332 @Callback  
333 protected QuotationCallback callback;  
334  
335 public double requestQuotation(String productCode, int quantity) {  
336     double price = getPrice(productCode, quantity);  
337     double discount = 0;  
338     if (quantity > 1000 && callback.getState().equals("FL")) {  
339         discount = 0.05;  
340     }  
341     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {  
342         discount += 0.05;  
343     }  
344     return price * (1-discount);  
345 }  
346 }  
347
```

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

```
351 public class ClientImpl implements ClientService, QuotationCallback {  
352     private QuotationService myService;  
353  
354     @Reference  
355     public void setMyService(QuotationService service) {  
356         myService = service;  
357     }  
358 }
```

```

359     }
360
361     public void aClientMethod() {
362         ...
363         double quote = myService.requestQuotation("AB123", 2000);
364         ...
365     }
366
367     public String getState() {
368         return "TX";
369     }
370     public String getZipCode() {
371         return "78746";
372     }
373     public String getCreditRating() {
374         return "AA";
375     }
376 }
377

```

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

382 6.2.2 Callback Instance Management

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

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

394 6.2.3 Implementing Multiple Bidirectional Interfaces

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

```

401
402 @Callback
403 protected MyService1Callback callback1;
404
405 @Callback
406 protected MyService2Callback callback2;
407

```

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

410 6.2.4 Accessing Callbacks

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

414

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

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

418

419 `@Callback`

420 `protected ServiceReference<MyCallback> callback;`

421

422 `public void someMethod() {`

423

424 `MyCallback myCallback = callback.getCallback(); ...`

425

426 `myCallback.receiveResult(theResult);`

427 `}`

428

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

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

435 `public void someMethod() {`

436

437 `MyCallback myCallback =`

438 `ComponentContext.getRequestContext().getCallback();`

439

440 `...`

441

442 `myCallback.receiveResult(theResult);`

443 `}`

444

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

447 `@Context`

448 `protected RequestContext requestContext;`

449

450 `void receiveResult(Object theResult) {`

451

452 `Object refParams =`

453 `requestContext.getServiceReference().getCallbackID();`

454 `...`

455 `}`

456

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

459 7 Policy Annotations for Java

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

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

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

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

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

486 7.1 General Intent Annotations

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

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

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

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

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

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

```
499     public static final String SCA_PREFIX=  
500         "{http://docs.oasis-open.org/ns/opencsa/sca/200712}";  
501     public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
502     public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
503
```

504 Notice that, by convention, qualified intents include the qualifier as part of the name of the
505 constant, separated by an underscore. These intent constants are defined in the file that defines

506 an annotation for the intent (annotations for intents, and the formal definition of these constants,
507 are covered in a following section).

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

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

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

512

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

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

```
515     package com.foo;
516
517     import static org.oasisopen.sca.annotation.Confidentiality.*;
518     import static org.oasisopen.sca.annotation.Reference;
519     import static org.oasisopen.sca.annotation.Requires;
520
521     public class Foo {
522         @Requires(CONFIDENTIALITY)
523         @Reference
524         public void setBar(Bar bar) {
525             ...
526         }
527     }
528
```

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

```
532     package com.foo;
533
534     import static org.oasisopen.sca.Constants.*;
535     import static org.oasisopen.sca.annotation.Reference;
536     import static org.oasisopen.sca.annotation.Requires;
537
538     public class Foo {
539         @Requires(SCA_PREFIX+"confidentiality")
540         @Reference
541         public void setBar(Bar bar) {
542             ...
543         }
544     }
545
```

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

```
547 '@Requires("'" QualifiedIntent "' (','" QualifiedIntent "'*)* ')
```

548 where

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

551

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

553 7.2 Specific Intent Annotations

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

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

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

```
565 @Integrity
```

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

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

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

571 The general form of specific intent annotations is:

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

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

```
574 Intent      ::= NCName  
575 qualifiers  ::= "" qualifier "" (',' qualifier "")*  
576 qualifier   ::= NCName ('.' qualifier)?  
577
```

578 7.2.1 How to Create Specific Intent Annotations

579 **SCA identifies annotations that correspond to intents by providing an @Intent annotation which**
580 **MUST be used in the definition of a specific intent annotation. [JCA70001]**

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

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

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

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

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

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

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

598 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
599 are set for the element to which the @confidentiality annotation is attached.

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

601 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
602 intent annotations are shown in [the section dealing with Security Interaction Policy](#).

603 7.3 Application of Intent Annotations

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

- 605 • Java class
- 606 • Java interface
- 607 • Method
- 608 • Field
- 609 • Constructor parameter

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

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

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

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

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

622 7.3.1 Inheritance And Annotation

623 The inheritance rules for annotations are consistent with the common annotation specification, JSR
624 250 [JSR-250]

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

```
627 package services.hello;  
628 import org.oasisopen.sca.annotation.Remotable;  
629 import org.oasisopen.sca.annotation.Integrity;  
630 import org.oasisopen.sca.annotation.Authentication;  
631  
632 @Integrity("transport")  
633 @Authentication  
634 public class HelloService {  
635     @Integrity  
636     @Authentication("message")  
637     public String hello(String message) {...}  
638  
639     @Integrity  
640     @Authentication("transport")  
641     public String helloThere() {...}  
642 }  
643  
644 package services.hello;  
645 import org.oasisopen.sca.annotation.Remotable;  
646 import org.oasisopen.sca.annotation.Confidentiality;  
647 import org.oasisopen.sca.annotation.Authentication;
```

```

648
649     @Confidentiality("message")
650     public class HelloChildService extends HelloService {
651         @Confidentiality("transport")
652         public String hello(String message) {...}
653         @Authentication
654         String helloWorld() {...}
655     }

```

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

657

658 The effective intent annotation on the **helloWorld** method of the **HelloChildService** is
659 Integrity("transport"), @Authentication, and @Confidentiality("message").

660 The effective intent annotation on the **hello** method of the **HelloChildService** is
661 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),

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

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

666

667 The listing below contains the equivalent declarative security interaction policy of the HelloService
668 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
669 Example 2a.

670

```

671     <?xml version="1.0" encoding="ASCII"?>
672     <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
673             name="HelloServiceComposite" >
674         <service name="HelloService" requires="integrity/transport
675             authentication">
676             ...
677         </service>
678         <service name="HelloChildService" requires="integrity/transport
679             authentication confidentiality/message">
680             ...
681         </service>
682         ...
683     </composite>
684     <component name="HelloServiceComponent">*
685         <implementation.java class="services.hello.HelloService"/>
686         <operation name="hello" requires="integrity
687             authentication/message"/>
688         <operation name="helloThere"
689             requires="integrity
690             authentication/transport"/>
691     </component>
692     <component name="HelloChildServiceComponent">*
693         <implementation.java
694             class="services.hello.HelloChildService" />
695         <operation name="hello"
696             requires="confidentiality/transport"/>
697         <operation name="helloThere" requires=" integrity/transport
698             authentication"/>
699         <operation name="helloWorld" requires="authentication"/>
700     </component>
701

```


702 ...
703
704 </composite>
705

706 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.
707

708 7.4 Relationship of Declarative And Annotated Intents

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

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

716 7.5 Policy Set Annotations

717 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies. For
718 example, a concrete policy is the specific encryption algorithm to use when encrypting messages
719 when using a specific communication protocol to link a reference to a service.
720

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

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

725

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

727 An example of the @PolicySets annotation:

728

```
729                   @Reference(name="helloService", required=true)  
730                   @PolicySets({ MY_NS + "WS_Encryption_Policy",  
731                                MY_NS + "WS_Authentication_Policy" })  
732                   public setHelloService(HelloService service) {  
733                    ...  
734                   }  
735
```

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

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

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

- 741 • Java class
- 742 • Java interface
- 743 • Method
- 744 • Field
- 745 • Constructor parameter

746 7.6 Security Policy Annotations

747 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy](#)
748 [Framework specification \[POLICY\]](#).

749 7.6.1 Security Interaction Policy

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

- 752 • @Integrity
- 753 • @Confidentiality
- 754 • @Authentication

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

- 756 • message
- 757 • transport

758 The formal definitions of the @Authentication, @Confidentiality and @Integrity annotations are
759 found in the sections [@Authentication](#), [@Confidentiality](#) and [@Integrity](#).

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

```
763
764 package services.hello;
765 //Interface for HelloService
766 public interface HelloService {
767     String hello(String helloMsg);
768 }
769
770 package services.client;
771 // Interface for ClientService
772 public interface ClientService {
773     public void clientMethod();
774 }
775
776 // Implementation class for ClientService
777 package services.client;
778
779 import services.hello.HelloService;
780 import org.oasisopen.sca.annotation.*;
781
782 @Service(ClientService.class)
783 public class ClientServiceImpl implements ClientService {
784
785     private HelloService helloService;
786
787     @Reference(name="helloService", required=true)
788     @Integrity("message")
789     @Authentication("message")
790     public void setHelloService(HelloService service) {
791         helloService = service;
792     }
793
794     public void clientMethod() {
795         String result = helloService.hello("Hello World!");
```

```
796     ...
797     }
798 }
799
```

800 Example 1. Usage of annotated intents on a reference.

801 7.6.2 Security Implementation Policy

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

- 805 • RunAs
806 Takes as a parameter a string which is the name of a Security role.
807 eg. @RunAs("Manager") Code marked with this annotation executes with the Security
808 permissions of the identified role.
- 810 • RolesAllowed
811 Takes as a parameter a single string or an array of strings which represent one or more
812 role names. When present, the implementation can only be accessed by principals whose
813 role corresponds to one of the role names listed in the @roles attribute. How role names
814 are mapped to security principals is implementation dependent (SCA does not define this).
815 eg. @RolesAllowed({"Manager", "Employee"})
- 817 • PermitAll
818 No parameters. When present, grants access to all roles.
- 820 • DenyAll
821 No parameters. When present, denies access to all roles.
- 823 • DeclareRoles
824 Takes as a parameter a string or an array of strings which identify one or more role names
825 that form the set of roles used by the implementation.
826 eg. @DeclareRoles({"Manager", "Employee", "Customer"})

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

828 For a full explanation of these intents, see [the Policy Framework specification \[POLICY\]](#).

829 7.6.2.1 Annotated Implementation Policy Example

830 The following is an example showing annotated security implementation policy:

```
831
832 package services.account;
833 @Remotable
834 public interface AccountService {
835     AccountReport getAccountReport(String customerID);
836     float fromUSDollarToCurrency(float value);
837 }
```

838
839 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,
840 plus the service references it makes and the settable properties that it has, along with a set of
841 implementation policy annotations:

```
842
843 package services.account;
```

```

844     import java.util.List;
845     import commonj.sdo.DataFactory;
846     import org.oasisopen.sca.annotation.Property;
847     import org.oasisopen.sca.annotation.Reference;
848     import org.oasisopen.sca.annotation.RolesAllowed;
849     import org.oasisopen.sca.annotation.RunAs;
850     import org.oasisopen.sca.annotation.PermitAll;
851     import services.accountdata.AccountDataService;
852     import services.accountdata.CheckingAccount;
853     import services.accountdata.SavingsAccount;
854     import services.accountdata.StockAccount;
855     import services.stockquote.StockQuoteService;
856     @RolesAllowed("customers")
857     @RunAs("accountants" )
858     public class AccountServiceImpl implements AccountService {
859
860         @Property
861         protected String currency = "USD";
862
863         @Reference
864         protected AccountDataService accountDataService;
865         @Reference
866         protected StockQuoteService stockQuoteService;
867
868         @RolesAllowed({"customers", "accountants"})
869         public AccountReport getAccountReport(String customerID) {
870
871             DataFactory dataFactory = DataFactory.INSTANCE;
872             AccountReport accountReport =
873                 (AccountReport)dataFactory.create(AccountReport.class);
874             List accountSummaries = accountReport.getAccountSummaries();
875
876             CheckingAccount checkingAccount =
877                 accountDataService.getCheckingAccount(customerID);
878             AccountSummary checkingAccountSummary =
879                 (AccountSummary)dataFactory.create(AccountSummary.class);
880             checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
881 );
882             checkingAccountSummary.setAccountType("checking");
883             checkingAccountSummary.setBalance(fromUSDollarToCurrency
884                 (checkingAccount.getBalance()));
885             accountSummaries.add(checkingAccountSummary);
886
887             SavingsAccount savingsAccount =
888                 accountDataService.getSavingsAccount(customerID);
889             AccountSummary savingsAccountSummary =
890                 (AccountSummary)dataFactory.create(AccountSummary.class);
891             savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
892             savingsAccountSummary.setAccountType("savings");
893             savingsAccountSummary.setBalance(fromUSDollarToCurrency
894                 (savingsAccount.getBalance()));
895             accountSummaries.add(savingsAccountSummary);
896
897             StockAccount stockAccount =
898                 accountDataService.getStockAccount(customerID);
899             AccountSummary stockAccountSummary =
900                 (AccountSummary)dataFactory.create(AccountSummary.class);
901             stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());

```

```

902         (AccountSummary)dataFactory.create(AccountSummary.class);
903     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
904     stockAccountSummary.setAccountType("stock");
905     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
906         stockAccount.getQuantity();
907     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
908     accountSummaries.add(stockAccountSummary);
909
910     return accountReport;
911 }
912
913 @PermitAll
914 public float fromUSDollarToCurrency(float value) {
915
916     if (currency.equals("USD")) return value;
917     if (currency.equals("EURO")) return value * 0.8f;
918     return 0.0f;
919 }
920 }

```

921 Example 3. Usage of annotated security implementation policy for the java language.

922 In this example, the implementation class as a whole is marked:

- 923 • @RolesAllowed("customers") - indicating that customers have access to the
- 924 implementation as a whole
- 925 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 926 permissions of accountants

927 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),
928 which indicates that this method can be called by both customers and accountants.

929 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
930 can be called by any role.

931 8 Java API

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

933 8.1 Component Context

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

```
935
936 package org.oasisopen.sca;
937
938 public interface ComponentContext {
939     String getURI();
940
941     <B> B getService(Class<B> businessInterface, String referenceName);
942
943     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
944                                             String referenceName);
945
946     <B> Collection<B> getServices(Class<B> businessInterface,
947                                String referenceName);
948
949     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
950                                                            businessInterface, String referenceName);
951
952     <B> ServiceReference<B> createSelfReference(Class<B>
953                                                businessInterface);
954
955     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
956                                                String serviceName);
957
958     <B> B getProperty(Class<B> type, String propertyName);
959
960     <B, R extends ServiceReference<B>> R cast(B target)
961         throws IllegalArgumentException;
962
963     RequestContext getRequestContext();
964
965
966 }
```

- 967
- 968 • **getURI()** - returns the absolute URI of the component within the SCA domain
 - 969 • **getService(Class businessInterface, String referenceName)** – Returns a proxy for
970 the reference defined by the current component. The getService() method takes as its
971 input arguments the Java type used to represent the target service on the client and the
972 name of the service reference. It returns an object providing access to the service. The
973 returned object implements the Java interface the service is typed with.
974 **ComponentContext.getService method MUST throw an IllegalArgumentException if the**
975 **reference identified by the referenceName parameter has multiplicity of 0..n or**
976 **1..n.[JCA80001]**
 - 977 • **getServiceReference(Class businessInterface, String referenceName)** – Returns a
978 ServiceReference defined by the current component. This method MUST throw an
979 IllegalArgumentException if the reference has multiplicity greater than one.

- 980 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
981 typed service proxies for a business interface type and a reference name.
- 982 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
983 list typed service references for a business interface type and a reference name.
- 984 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
985 be used to invoke this component over the designated service.
- 986 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
987 ServiceReference that can be used to invoke this component over the designated service.
988 Service name explicitly declares the service name to invoke
- 989 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
990 property defined by this component.
- 991 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
992 there is no current request or if the context is unavailable. **The**
993 **ComponentContext.getRequestContext** method **MUST return non-null when invoked during**
994 **the execution of a Java business method for a service operation or a callback operation, on**
995 **the same thread that the SCA runtime provided, and MUST return null in all other cases.**
996 **[JCA80002]**
- 997 • **cast(B target)** - Casts a type-safe reference to a ServiceReference

998 A component can access its component context by defining a field or setter method typed by
999 **org.oasisopen.sca.ComponentContext** and annotated with **@Context**. To access the target
1000 service, the component uses **ComponentContext.getService(..)**.

1001 The following shows an example of component context usage in a Java class using the @Context
1002 annotation.

```
1003 private ComponentContext componentContext;
1004
1005 @Context
1006 public void setContext(ComponentContext context) {
1007     componentContext = context;
1008 }
1009
1010 public void doSomething() {
1011     HelloWorld service =
1012         componentContext.getService(HelloWorld.class, "HelloWorldComponent");
1013     service.hello("hello");
1014 }
1015
```

1016 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
1017 component in an SCA domain. How the non-SCA client code obtains a reference to a
1018 ComponentContext is runtime specific.

Comment [ME2]: Need to reexamine this in the light of Issue 1 resolution

1019 8.2 Request Context

1020 The following shows the **RequestContext** interface:

```
1021
1022 package org.oasisopen.sca;
1023
1024 import javax.security.auth.Subject;
1025
1026 public interface RequestContext {
1027
1028     Subject getSecuritySubject();
1029
```

```

1030     String getServiceName();
1031     <CB> ServiceReference<CB> getCallbackReference();
1032     <CB> CB getCallback();
1033     <B> ServiceReference<B> getServiceReference();
1034
1035 }
1036

```

1037 The RequestContext interface has the following methods:

- 1038 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 1039 • **getServiceName()** – Returns the name of the service on the Java implementation the
1040 request came in on
- 1041 • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1042 caller. This method returns null when called for a service request whose interface is not
1043 bidirectional or when called for a callback request.
- 1044 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1045 getCallbackReference() method, this method returns null when called for a service request
1046 whose interface is not bidirectional or when called for a callback request.
- 1047 • **getServiceReference()** – When invoked during the execution of a service operation, the
1048 getServiceReference method MUST return a ServiceReference that represents the service
1049 that was invoked. When invoked during the execution of a callback operation, the
1050 getServiceReference method MUST return a ServiceReference that represents the callback
1051 that was invoked. [JCA80003]

Comment [ME3]: Need a reference to JAAS here

Comment [ME4]: What happens if there is no JAAS subject?

1052 8.3 ServiceReference

1053 ServiceReferences can be injected using the @Reference annotation on a field, a setter method,
1054 or constructor parameter taking the type ServiceReference. The detailed description of the usage
1055 of these methods is described in the section on Asynchronous Programming in this document.

1056 The following Java code defines the **ServiceReference** interface:

```

1057 package org.oasisopen.sca;
1058
1059 public interface ServiceReference<B> extends java.io.Serializable {
1060
1061     B getService();
1062     Class<B> getBusinessInterface();
1063 }
1064

```

1065 The ServiceReference interface has the following methods:

- 1066 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
1067 returned is guaranteed to implement the business interface for this reference. The value
1068 returned is a proxy to the target that implements the business interface associated with this
1069 reference.
- 1070 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1071 this reference.

1072 8.4 ServiceRuntimeException

1073 The following snippet shows the **ServiceRuntimeException**.

```

1074
1075 package org.oasisopen.sca;
1076

```



```
1077     public class ServiceRuntimeException extends RuntimeException {
1078         ...
1079     }
1080
```

1081 This exception signals problems in the management of SCA component execution.

1082 8.5 ServiceUnavailableException

1083 The following snippet shows the *ServiceUnavailableException*.

```
1084     package org.oasisopen.sca;
1085
1086     public class ServiceUnavailableException extends ServiceRuntimeException {
1087         ...
1088     }
1089
1090
```

1091 This exception signals problems in the interaction with remote services. These are exceptions
1092 that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException
1093 that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since
1094 it most likely requires human intervention

1095 8.6 InvalidServiceException

1096 The following snippet shows the *InvalidServiceException*.

```
1097     package org.oasisopen.sca;
1098
1099     public class InvalidServiceException extends ServiceRuntimeException {
1100         ...
1101     }
1102
1103
```

1104 This exception signals that the ServiceReference is no longer valid. This can happen when the
1105 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1106 be resolved by retrying the operation and will most likely require human intervention.

1107 8.7 Constants

1108 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java
1109 APIs and Annotations. The following snippet shows the Constants interface:

```
1110     package org.oasisopen.sca;
1111
1112     public interface Constants {
1113         String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1114         String SCA_PREFIX = "{"+SCA_NS+"}";
1115     }
1116
```

1117 8.8 SCAClient Interface

1118 The SCAClient interface can be used by client code to obtain a proxy reference object for a service
1119 within an SCA Domain, through which the client code can invoke operations of that service. This
1120 is particularly useful for client code that is running outside the SCA Domain containing the target
1121 service, for example where the code is "unmanaged" and is not running under an SCA runtime.

1122 The following shows the *SCAClient* interface:

```
1123     package org.oasisopen.sca.client;
```

```

1124 public interface SCAClient {
1125
1126     <T> T getService(Class<T> interfaze,
1127                     String serviceURI,
1128                     URI domainURI)
1129     throws NoSuchServiceException, NoSuchDomainException;
1130
1131 }

```

1132 getService method:

1133 Obtains a proxy reference object for a specified target service in a specified SCA Domain.

1134 Returns:

- 1135 • proxy object which implements the business interface T
- 1136 Invocations of a business method of the proxy causes the invocation of the corresponding
- 1137 operation of the target service .

1138 Parameters:

- 1139 • interfaze - a Java interface class which is the business interface of the target service
- 1140 • serviceURI - a String containing the relative URI of the target service within its SCA
- 1141 Domain.
- 1142 Takes the form componentName/serviceName or can also take the extended form
- 1143 componentName/serviceName/bindingName to use a specific binding of the target service
- 1144 • domainURI - a URI for the SCA Domain containing the target service

1145 Exceptions:

- 1146 • NoSuchServiceException - thrown if a service with the relative URI serviceURI and a
- 1147 business interface which matches **interfaze** cannot be found in the Domain identified by
- 1148 **domainURI**
- 1149 • NoSuchDomainException - thrown if the Domain identified by domainURI cannot be
- 1150 found

1151

1152 **8.9 SCAClientFactory Class**

1153 The SCAClientFactory class provides the means for client code to obtain an object which

1154 implements the SCAClient interface which is used in turn to obtain a proxy reference object to a

1155 service within an SCA Domain.

1156 The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods

1157 which the client can invoke in order to obtain a object implementing the SCAClient interface.

1158 The SCAClientFactory class is as follows:

```

1159 public abstract class SCAClientFactory {
1160
1161     private static SCAClientFactory defaultFactory;
1162
1163     public static SCAClient newInstance() {
1164         return newInstance(null, null);
1165     }
1166
1167     public static SCAClient newInstance(Properties properties) {
1168         return newInstance(properties, null);
1169     }
1170
1171     public static SCAClient newInstance(ClassLoader classLoader) {
1172         return newInstance(null, classLoader);

```

```

1173     }
1174
1175     public static SCAClient newInstance(Properties properties,
1176                                      ClassLoader classLoader) {
1177         final SCAClientFactory factory;
1178         if(defaultFactory == null) {
1179             factory = SCAClientFactoryFinder.find(properties,
1180                                                  classLoader);
1181         } else {
1182             factory = defaultFactory;
1183         }
1184         return factory.createSCAClient();
1185     }
1186
1187     protected abstract SCAClient createSCAClient();
1188 }

```

1189 **newInstance() method:**

1190 Obtains a object implementing the SCAClient interface.

1191 Returns:

- 1192 • **object** which implements the SCAClient interface

1193 Parameters:

- 1194 • **none**

1195 Exceptions:

- 1196 • **none**

1197

1198 **newInstance(Properties) method:**

1199 Obtains a object implementing the SCAClient interface, using a specified set of properties.

1200 Returns:

- 1201 • **object** which implements the SCAClient interface

1202 Parameters:

- 1203 • **properties** - a set of Properties that can be used when creating the object which
- 1204 implements the SCAClient interface.

1205 Exceptions:

1206 **none**

1207

1208 **newInstance(Classloader) method:**

1209 Obtains a object implementing the SCAClient interface using a specified classloader.

1210 Returns:

- 1211 • **object** which implements the SCAClient interface

1212 Parameters:

- 1213 • **classLoader** - a ClassLoader to use when creating the object which implements the
- 1214 SCAClient interface.

1215 Exceptions:

1216 **none**

1217

1218 **[newInstance\(Properties, Classloader\) method:](#)**
1219 Obtains a object implementing the SCAClient interface using a specified set of properties and a
1220 specified classloader.
1221 Returns:
1222

- **object** which implements the SCAClient interface

1223 Parameters:
1224

- **properties** - a set of Properties that can be used when creating the object which
1225 implements the SCAClient interface.
- **classLoader** - a ClassLoader to use when creating the object which implements the
1226 SCAClient interface.

1227
1228 Exceptions:
1229 **none**
1230

1231 **8.10 NoSuchDomainException**

1232 The following shows the NoSuchDomainException:

```
1233 package org.oasisopen.sca;  
1234  
1235 public class NoSuchDomainException extends Exception {  
1236     ...  
1237 }
```

1238 This exception indicates that the Domain specified could not be found.

1239 **8.11 NoSuchServiceException**

1240 The following shows the NoSuchServiceException:

```
1241 package org.oasisopen.sca;  
1242  
1243 public class NoSuchServiceException extends Exception {  
1244     ...  
1245 }
```

1246 This exception indicates that the service specified could not be found.

1247 9 Java Annotations

1248 This section provides definitions of all the Java annotations which apply to SCA.

1249 This specification places constraints on some annotations that are not detectable by a Java
1250 compiler. For example, the definition of the `@Property` and `@Reference` annotations indicate that
1251 they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to
1252 constructor parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if
1253 an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the
1254 invalid implementation code. [JCA90001]

1255 SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an
1256 SCA annotation on a static method or a static field of an implementation class and the SCA
1257 runtime MUST NOT instantiate such an implementation class. [JCA90002]

1258 9.1 @AllowsPassByReference

1259 The following Java code defines the `@AllowsPassByReference` annotation:

1260

```
1261 package org.oasisopen.sca.annotation;  
1262  
1263 import static java.lang.annotation.ElementType.TYPE;  
1264 import static java.lang.annotation.ElementType.METHOD;  
1265 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1266 import java.lang.annotation.Retention;  
1267 import java.lang.annotation.Target;  
1268  
1269 @Target({TYPE, METHOD})  
1270 @Retention(RUNTIME)  
1271 public interface AllowsPassByReference {  
1272  
1273 }  
1274
```

1275 The `@AllowsPassByReference` annotation is used on implementations of remotable interfaces to
1276 indicate that interactions with the service from a client within the same address space are allowed
1277 to use pass by reference data exchange semantics. The implementation promises that its by-value
1278 semantics will be maintained even if the parameters and return values are actually passed by-
1279 reference. This means that the service will not modify any operation input parameter or return
1280 value, even after returning from the operation. Either a whole class implementing a remotable
1281 service or an individual remotable service method implementation can be annotated using the
1282 `@AllowsPassByReference` annotation.

1283 `@AllowsPassByReference` has no attributes

1284 The following snippet shows a sample where `@AllowsPassByReference` is defined for the
1285 implementation of a service method on the Java component implementation class.

1286

```
1287 @AllowsPassByReference  
1288 public String hello(String message) {  
1289     ...  
1290 }
```

1291 9.2 @Authentication

1292 The following Java code defines the `@Authentication` annotation:

```

1293
1294 package org.oasisopen.sca.annotation;
1295
1296 import static java.lang.annotation.ElementType.FIELD;
1297 import static java.lang.annotation.ElementType.METHOD;
1298 import static java.lang.annotation.ElementType.PARAMETER;
1299 import static java.lang.annotation.ElementType.TYPE;
1300 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1301 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1302
1303 import java.lang.annotation.Inherited;
1304 import java.lang.annotation.Retention;
1305 import java.lang.annotation.Target;
1306
1307 @Inherited
1308 @Target({TYPE, FIELD, METHOD, PARAMETER})
1309 @Retention(RUNTIME)
1310 @Intent(Authentication.AUTHENTICATION)
1311 public @interface Authentication {
1312     String AUTHENTICATION = SCA_PREFIX + "authentication";
1313     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
1314     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";
1315
1316     /**
1317      * List of authentication qualifiers (such as "message"
1318      * or "transport").
1319      *
1320      * @return authentication qualifiers
1321      */
1322     @Qualifier
1323     String[] value() default "";
1324 }

```

1325 The **@Authentication** annotation is used to indicate that the invocation requires authentication.
1326 See the [section on Application of Intent Annotations](#) for samples and details.

1327 9.3 @Callback

1328 The following Java code defines the **@Callback** annotation:

```

1329
1330 package org.oasisopen.sca.annotation;
1331
1332 import static java.lang.annotation.ElementType.TYPE;
1333 import static java.lang.annotation.ElementType.METHOD;
1334 import static java.lang.annotation.ElementType.FIELD;
1335 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1336 import java.lang.annotation.Retention;
1337 import java.lang.annotation.Target;
1338
1339 @Target(TYPE, METHOD, FIELD)
1340 @Retention(RUNTIME)
1341 public @interface Callback {
1342
1343     Class<?> value() default Void.class;
1344 }
1345
1346

```

1347 The @Callback annotation is used to annotate a service interface with a callback interface by
1348 specifying the Java class object of the callback interface as an attribute.

1349 The @Callback annotation has the following attribute:

- 1350 • **value** – the name of a Java class file containing the callback interface

1351

1352 The @Callback annotation can also be used to annotate a method or a field of an SCA
1353 implementation class, in order to have a callback object injected. When used to annotate a
1354 method or a field of an implementation class for injection of a callback object, the @Callback
1355 annotation MUST NOT specify any attributes. [JCA90046]

1356 An example use of the @Callback annotation to declare a callback interface follows:

```
1357 package somepackage;  
1358 import org.oasisopen.sca.annotation.Callback;  
1359 import org.oasisopen.sca.annotation.Remotable;  
1360 @Remotable  
1361 @Callback(MyServiceCallback.class)  
1362 public interface MyService {  
1363     void someMethod(String arg);  
1364 }  
1365  
1366 @Remotable  
1367 public interface MyServiceCallback {  
1368     void receiveResult(String result);  
1369 }  
1370  
1371 }  
1372
```

1372

1373 In this example, the implied component type is:

```
1374 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
1375     <service name="MyService">  
1376         <interface.java interface="somepackage.MyService"  
1377             callbackInterface="somepackage.MyServiceCallback"/>  
1378     </service>  
1379 </componentType>
```

1381 9.4 @ComponentName

1382 The following Java code defines the @ComponentName annotation:

1383

```
1384 package org.oasisopen.sca.annotation;  
1385  
1386 import static java.lang.annotation.ElementType.METHOD;  
1387 import static java.lang.annotation.ElementType.FIELD;  
1388 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1389 import java.lang.annotation.Retention;  
1390 import java.lang.annotation.Target;  
1391  
1392 @Target({METHOD, FIELD})  
1393 @Retention(RUNTIME)  
1394 public @interface ComponentName {  
1395  
1396 }  
1397
```

1397

1398 The @ComponentName annotation is used to denote a Java class field or setter method that is
1399 used to inject the component name.

1400 The following snippet shows a component name field definition sample.

```
1401  
1402 @ComponentName  
1403 private String componentName;  
1404
```

1405 The following snippet shows a component name setter method sample.

```
1406  
1407 @ComponentName  
1408 public void setComponentName(String name) {  
1409     //...  
1410 }
```

1411 9.5 @Confidentiality

1412 The following Java code defines the **@Confidentiality** annotation:

```
1413  
1414 package org.oasisopen.sca.annotations;  
1415  
1416 import static java.lang.annotation.ElementType.FIELD;  
1417 import static java.lang.annotation.ElementType.METHOD;  
1418 import static java.lang.annotation.ElementType.PARAMETER;  
1419 import static java.lang.annotation.ElementType.TYPE;  
1420 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1421 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
1422  
1423 import java.lang.annotation.Inherited;  
1424 import java.lang.annotation.Retention;  
1425 import java.lang.annotation.Target;  
1426  
1427 @Inherited  
1428 @Target({TYPE, FIELD, METHOD, PARAMETER})  
1429 @Retention(RUNTIME)  
1430 @Intent(Confidentiality.CONFIDENTIALITY)  
1431 public @interface Confidentiality {  
1432     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
1433     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
1434     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";  
1435  
1436     /**  
1437      * List of confidentiality qualifiers such as "message" or  
1438      * "transport".  
1439      *  
1440      * @return confidentiality qualifiers  
1441      */  
1442     @Qualifier  
1443     String[] value() default "";  
1444 }
```

1445 The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.

1446 See the [section on Application of Intent Annotations](#) for samples and details.

1447 9.6 @Constructor

1448 The following Java code defines the **@Constructor** annotation:

```
1449 package org.oasisopen.sca.annotation;
1450
1451 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1452 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1453 import java.lang.annotation.Retention;
1454 import java.lang.annotation.Target;
1455
1456 @Target({CONSTRUCTOR})
1457 @Retention(RUNTIME)
1458 public @interface Constructor { }
```

1461 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1462 Java component implementation. **If a constructor of an implementation class is annotated with
1463 @Constructor and the constructor has parameters, each of these parameters MUST have either a
1464 @Property annotation or a @Reference annotation. [JCA90003]**

1465 The following snippet shows a sample for the @Constructor annotation.

```
1466
1467 public class HelloServiceImpl implements HelloService {
1468
1469     public HelloServiceImpl(){
1470         ...
1471     }
1472
1473     @Constructor
1474     public HelloServiceImpl(@Property(name="someProperty")
1475                             String someProperty ){
1476         ...
1477     }
1478
1479     public String hello(String message) {
1480         ...
1481     }
1482 }
```

Comment [ME5]: There also needs to be a normative statement that at most 1 constructor can be annotated with @Constructor

1483 9.7 @Context

1484 The following Java code defines the **@Context** annotation:

```
1485
1486 package org.oasisopen.sca.annotation;
1487
1488 import static java.lang.annotation.ElementType.METHOD;
1489 import static java.lang.annotation.ElementType.FIELD;
1490 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1491 import java.lang.annotation.Retention;
1492 import java.lang.annotation.Target;
1493
1494 @Target({METHOD, FIELD})
1495 @Retention(RUNTIME)
1496 public @interface Context {
1497 }
```

1498 }
1499

1500 The @Context annotation is used to denote a Java class field or a setter method that is used to
1501 inject a composite context for the component. The type of context to be injected is defined by the
1502 type of the Java class field or type of the setter method input argument; the type is either
1503 **ComponentContext** or **RequestContext**.

1504 The @Context annotation has no attributes.

1505 The following snippet shows a ComponentContext field definition sample.

```
1506  
1507 @Context  
1508 protected ComponentContext context;  
1509
```

1510 The following snippet shows a RequestContext field definition sample.

```
1511  
1512 @Context  
1513 protected RequestContext context;
```

1514 9.8 @Destroy

1515 The following Java code defines the **@Destroy** annotation:

```
1516  
1517 package org.oasisopen.sca.annotation;  
1518  
1519 import static java.lang.annotation.ElementType.METHOD;  
1520 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1521 import java.lang.annotation.Retention;  
1522 import java.lang.annotation.Target;  
1523  
1524 @Target(METHOD)  
1525 @Retention(RUNTIME)  
1526 public @interface Destroy {  
1527  
1528 }  
1529
```

1530 The @Destroy annotation is used to denote a single Java class method that will be called when the
1531 scope defined for the implementation class ends. A method annotated with @Destroy MAY have
1532 any access modifier and MUST have a void return type and no arguments. [JCA90004]

1533 If there is a method annotated with @Destroy that matches the criteria for the annotation, the
1534 SCA runtime MUST call the annotated method when the scope defined for the implementation
1535 class ends. [JCA90005]

1536 The following snippet shows a sample for a destroy method definition.

```
1537  
1538 @Destroy  
1539 public void myDestroyMethod() {  
1540     ...  
1541 }
```

1542 9.9 @EagerInit

1543 The following Java code defines the **@EagerInit** annotation:

```

1544
1545 package org.oasisopen.sca.annotation;
1546
1547 import static java.lang.annotation.ElementType.TYPE;
1548 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1549 import java.lang.annotation.Retention;
1550 import java.lang.annotation.Target;
1551
1552 @Target(TYPE)
1553 @Retention(RUNTIME)
1554 public @interface EagerInit {
1555
1556 }
1557

```

1558 The **@EagerInit** annotation is used to mark the Java class of a COMPOSITE scoped
1559 implementation for eager initialization. When marked for eager initialization with an @EagerInit
1560 annotation, the composite scoped instance MUST be created when its containing component is
1561 started. [JCA90007]

1562 9.10 @Init

1563 The following Java code defines the **@Init** annotation:

```

1564
1565 package org.oasisopen.sca.annotation;
1566
1567 import static java.lang.annotation.ElementType.METHOD;
1568 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1569 import java.lang.annotation.Retention;
1570 import java.lang.annotation.Target;
1571
1572 @Target(METHOD)
1573 @Retention(RUNTIME)
1574 public @interface Init {
1575
1576 }
1577
1578

```

1579 The @Init annotation is used to denote a single Java class method that is called when the scope
1580 defined for the implementation class starts. A method marked with the @Init annotation MAY
1581 have any access modifier and MUST have a void return type and no arguments. [JCA90008]

1582 If there is a method annotated with @Init that matches the criteria for the annotation, the SCA
1583 runtime MUST call the annotated method after all property and reference injection is complete.
1584 [JCA90009]

1585 The following snippet shows an example of an init method definition.

```

1586
1587 @Init
1588 public void myInitMethod() {
1589     ...
1590 }

```

1591 9.11 @Integrity

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

1593

```

1594 package org.oasisopen.sca.annotation;
1595
1596 import static java.lang.annotation.ElementType.FIELD;
1597 import static java.lang.annotation.ElementType.METHOD;
1598 import static java.lang.annotation.ElementType.PARAMETER;
1599 import static java.lang.annotation.ElementType.TYPE;
1600 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1601 import static org.oasisopen.Constants.SCA_PREFIX;
1602
1603 import java.lang.annotation.Inherited;
1604 import java.lang.annotation.Retention;
1605 import java.lang.annotation.Target;
1606
1607 @Inherited
1608 @Target({TYPE, FIELD, METHOD, PARAMETER})
1609 @Retention(RUNTIME)
1610 @Intent(Integrity.INTEGRITY)
1611 public @interface Integrity {
1612     String INTEGRITY = SCA_PREFIX + "integrity";
1613     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
1614     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
1615
1616     /**
1617      * List of integrity qualifiers (such as "message" or "transport").
1618      *
1619      * @return integrity qualifiers
1620      */
1621     @Qualifier
1622     String[] value() default "";
1623 }
1624

```

1625 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no
1626 tampering of the messages between client and service).

1627 See the [section on Application of Intent Annotations](#) for samples and details.

1628 9.12 @Intent

1629 The following Java code defines the **@Intent** annotation:

```

1630 package org.oasisopen.sca.annotation;
1631
1632 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1633 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1634 import java.lang.annotation.Retention;
1635 import java.lang.annotation.Target;
1636
1637 @Target({ANNOTATION_TYPE})
1638 @Retention(RUNTIME)
1639 public @interface Intent {
1640     /**
1641      * The qualified name of the intent, in the form defined by
1642      * {@link javax.xml.namespace.QName#toString}.
1643      * @return the qualified name of the intent
1644      */
1645     String value() default "";
1646
1647     /**
1648

```

```

1649     * The XML namespace for the intent.
1650     * @return the XML namespace for the intent
1651     */
1652     String targetNamespace() default "";
1653
1654     /**
1655     * The name of the intent within its namespace.
1656     * @return name of the intent within its namespace
1657     */
1658     String localPart() default "";
1659 }
1660

```

1661 The @Intent annotation is used for the creation of new annotations for specific intents. It is not
 1662 expected that the @Intent annotation will be used in application code.

1663 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
 1664 define new intent annotations.

1665 9.13 @OneWay

1666 The following Java code defines the **@OneWay** annotation:

```

1667
1668 package org.oasisopen.sca.annotation;
1669
1670 import static java.lang.annotation.ElementType.METHOD;
1671 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1672 import java.lang.annotation.Retention;
1673 import java.lang.annotation.Target;
1674
1675 @Target(METHOD)
1676 @Retention(RUNTIME)
1677 public @interface OneWay {
1678
1679 }
1680
1681

```

1682 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
 1683 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
 1684 [Programming](#).

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

1685 The @OneWay annotation has no attributes.

1686 The following snippet shows the use of the @OneWay annotation on an interface.

```

1687 package services.hello;
1688
1689 import org.oasisopen.sca.annotation.OneWay;
1690
1691 public interface HelloService {
1692     @OneWay
1693     void hello(String name);
1694 }

```

1695 9.14 @PolicySets

1696 The following Java code defines the **@PolicySets** annotation:

```

1697 package org.oasisopen.sca.annotation;
1698

```

```

1699
1700 import static java.lang.annotation.ElementType.FIELD;
1701 import static java.lang.annotation.ElementType.METHOD;
1702 import static java.lang.annotation.ElementType.PARAMETER;
1703 import static java.lang.annotation.ElementType.TYPE;
1704 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1705
1706 import java.lang.annotation.Retention;
1707 import java.lang.annotation.Target;
1708
1709 @Target({TYPE, FIELD, METHOD, PARAMETER})
1710 @Retention(RUNTIME)
1711 public @interface PolicySets {
1712     /**
1713      * Returns the policy sets to be applied.
1714      *
1715      * @return the policy sets to be applied
1716      */
1717     String[] value() default "";
1718 }
1719

```

1720 The **@PolicySets** annotation is used to attach one or more SCA Policy Sets to a Java
1721 implementation class or to one of its subelements.

1722 See the [section "Policy Set Annotations"](#) for details and samples.

1723 9.15 @Property

1724 The following Java code defines the **@Property** annotation:

```

1725 package org.oasisopen.sca.annotation;
1726
1727 import static java.lang.annotation.ElementType.METHOD;
1728 import static java.lang.annotation.ElementType.FIELD;
1729 import static java.lang.annotation.ElementType.PARAMETER;
1730 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1731 import java.lang.annotation.Retention;
1732 import java.lang.annotation.Target;
1733
1734 @Target({METHOD, FIELD, PARAMETER})
1735 @Retention(RUNTIME)
1736 public @interface Property {
1737
1738     String name() default "";
1739     boolean required() default true;
1740 }
1741

```

1742 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1743 parameter that is used to inject an SCA property value. The type of the property injected, which
1744 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1745 the type of the input parameter of the setter method or constructor.

1746 The @Property annotation can be used on fields, on setter methods or on a constructor method
1747 parameter. However, **the @Property annotation MUST NOT be used on a class field that is declared
1748 as final.** [JCA90011]

1749 Properties can also be injected via setter methods even when the @Property annotation is not
1750 present. However, **the @Property annotation MUST be used in order to inject a property onto a
1751 non-public field.** [JCA90012] In the case where there is no @Property annotation, the name of the
1752 property is the same as the name of the field or setter.

1753 Where there is both a setter method and a field for a property, the setter method is used.

1754 The @Property annotation has the following attributes:

- 1755 • **name (optional)** – the name of the property. For a field annotation, the default is the
1756 name of the field of the Java class. For a setter method annotation, the default is the
1757 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1758 @Property annotation applied to a constructor parameter, there is no default value for the
1759 name attribute and the name attribute MUST be present. [JCA90013]
- 1760 • **required (optional)** – a boolean value which specifies whether injection of the property
1761 value is required or not, where true means injection is required and false means injection
1762 is not required. Defaults to true. For a @Property annotation applied to a constructor
1763 parameter, the required attribute MUST have the value true. [JCA90014]

1764

1765 The following snippet shows a property field definition sample.

1766

```
1767 @Property(name="currency", required=true)  
1768 protected String currency;
```

1769

1770 The following snippet shows a property setter sample

1771

```
1772 @Property(name="currency", required=true)  
1773 public void setCurrency( String theCurrency ) {  
1774     ....  
1775 }
```

1776

1777 For a @Property annotation, if the the type of the Java class field or the type of the input
1778 parameter of the setter method or constructor is defined as an array or as any type that extends
1779 or implements java.util.Collection, then the SCA runtime MUST introspect the component type of
1780 the implementation with a <property/> element with a @many attribute set to true, otherwise
1781 @many MUST be set to false. [JCA90047]

1782 The following snippet shows the definition of a configuration property using the @Property
1783 annotation for a collection.

```
1784 ...  
1785 private List<String> helloConfigurationProperty;  
1786  
1787 @Property(required=true)  
1788 public void setHelloConfigurationProperty(List<String> property) {  
1789     helloConfigurationProperty = property;  
1790 }  
1791 ...
```

1792 9.16 @Qualifier

1793 The following Java code defines the @Qualifier annotation:

1794

```
1795 package org.oasisopen.sca.annotation;  
1796  
1797 import static java.lang.annotation.ElementType.METHOD;  
1798 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1799
```

```

1800     import java.lang.annotation.Retention;
1801     import java.lang.annotation.Target;
1802
1803     @Target(METHOD)
1804     @Retention(RUNTIME)
1805     public @interface Qualifier {
1806     }
1807

```

1808 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,
1809 defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the
1810 intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the
1811 intent has qualifiers. [JCA90015]

1812 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
1813 define new intent annotations.

1814 9.17 @Reference

1815 The following Java code defines the **@Reference** annotation:

```

1816
1817     package org.oasisopen.sca.annotation;
1818
1819     import static java.lang.annotation.ElementType.METHOD;
1820     import static java.lang.annotation.ElementType.FIELD;
1821     import static java.lang.annotation.ElementType.PARAMETER;
1822     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1823     import java.lang.annotation.Retention;
1824     import java.lang.annotation.Target;
1825     @Target({METHOD, FIELD, PARAMETER})
1826     @Retention(RUNTIME)
1827     public @interface Reference {
1828
1829         String name() default "";
1830         boolean required() default true;
1831     }
1832

```

1833 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1834 constructor parameter that is used to inject a service that resolves the reference. The interface of
1835 the service injected is defined by the type of the Java class field or the type of the input parameter
1836 of the setter method or constructor.

1837 The @Reference annotation MUST NOT be used on a class field that is declared as final.
1838 [JCA90016]

1839 References can also be injected via setter methods even when the @Reference annotation is not
1840 present. However, the @Reference annotation MUST be used in order to inject a reference onto a
1841 non-public field. [JCA90017] In the case where there is no @Reference annotation, the name of
1842 the reference is the same as the name of the field or setter.

1843 Where there is both a setter method and a field for a reference, the setter method is used.

1844 The @Reference annotation has the following attributes:

- 1845 • **name : String (optional)** – the name of the reference. For a field annotation, the default is
1846 the name of the field of the Java class. For a setter method annotation, the default is the
1847 JavaBeans property name corresponding to the setter method name. For a @Reference
1848 annotation applied to a constructor parameter, there is no default for the name attribute
1849 and the name attribute MUST be present. [JCA90018]

- 1850
- **required (optional)** – a boolean value which specifies whether injection of the service reference is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true. [JCA90019]
- 1851
1852
1853

1854

1855 The following snippet shows a reference field definition sample.

1856

```
1857 @Reference(name="stockQuote", required=true)
1858 protected StockQuoteService stockQuote;
```

1859

1860 The following snippet shows a reference setter sample

1861

```
1862 @Reference(name="stockQuote", required=true)
1863 public void setStockQuote( StockQuoteService theSQService ) {
1864     ...
1865 }
```

1866

1867 The following fragment from a component implementation shows a sample of a service reference using the @Reference annotation. The name of the reference is "helloService" and its type is HelloService. The clientMethod() calls the "hello" operation of the service referenced by the helloService reference.

1871

```
1872 package services.hello;
1873
1874 private HelloService helloService;
1875
1876 @Reference(name="helloService", required=true)
1877 public setHelloService(HelloService service) {
1878     helloService = service;
1879 }
1880
1881 public void clientMethod() {
1882     String result = helloService.hello("Hello World!");
1883     ...
1884 }
1885
```

1886 The presence of a @Reference annotation is reflected in the componentType information that the runtime generates through reflection on the implementation class. The following snippet shows the component type for the above component implementation fragment.

1889

```
1890 <?xml version="1.0" encoding="ASCII"?>
1891 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1892     <!-- Any services offered by the component would be listed here -->
1893     <reference name="helloService" multiplicity="1..1">
1894         <interface.java interface="services.hello.HelloService"/>
1895     </reference>
1896 </componentType>
```

1897

1900 If the type of a reference is not an array or any type that extends or implements
1901 java.util.Collection, then the SCA runtime MUST introspect the component type of the
1902 implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference
1903 annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation
1904 required attribute is true. [JCA90020]

1905 If the type of a reference is defined as an array or as any type that extends or implements
1906 java.util.Collection, then the SCA runtime MUST introspect the component type of the
1907 implementation with a <reference/> element with @multiplicity=0..n if the @Reference
1908 annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation
1909 required attribute is true. [JCA90021]

1910 The following fragment from a component implementation shows a sample of a service reference
1911 definition using the @Reference annotation on a java.util.List. The name of the reference is
1912 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1913 services referenced by the helloServices reference. In this case, at least one HelloService needs
1914 to be present, so **required** is true.

```
1915 @Reference(name="helloServices", required=true)  
1916 protected List<HelloService> helloServices;  
1917  
1918 public void clientMethod() {  
1919     ...  
1920     for (int index = 0; index < helloServices.size(); index++) {  
1921         HelloService helloService =  
1922             (HelloService)helloServices.get(index);  
1923         String result = helloService.hello("Hello World!");  
1924     }  
1925     ...  
1926 }  
1927 }  
1928 }  
1929 }
```

1930 The following snippet shows the XML representation of the component type reflected from for the
1931 former component implementation fragment. There is no need to author this component type in
1932 this case since it can be reflected from the Java class.

```
1933  
1934 <?xml version="1.0" encoding="ASCII"?>  
1935 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1936     <!-- Any services offered by the component would be listed here -->  
1937     <reference name="helloServices" multiplicity="1..n">  
1938         <interface.java interface="services.hello.HelloService"/>  
1939     </reference>  
1940 </componentType>  
1941  
1942 </componentType>
```

1943 An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by
1944 the SCA runtime as null [JCA90022] An unwired reference with a multiplicity of 0..n MUST be
1945 presented to the implementation code by the SCA runtime as an empty array or empty collection
1946 [JCA90023]
1947

1948 9.17.1 Reinjection

1949 References MAY be reinjected by an SCA runtime after the initial creation of a component if the
1950 reference target changes due to a change in wiring that has occurred since the component was
1951 initialized. [JCA90024]

1952 In order for reinjection to occur, the following MUST be true:

1953 1. The component MUST NOT be STATELESS scoped.

1954 2. The reference MUST use either field-based injection or setter injection. References that are

1955 injected through constructor injection MUST NOT be changed.

1956 [JCA90025]

1957 Setter injection allows for code in the setter method to perform processing in reaction to a change.

1958 If a reference target changes and the reference is not reinjected, the reference MUST continue to

1959 work as if the reference target was not changed. [JCA90026]

1960 If an operation is called on a reference where the target of that reference has been undeployed,

1961 the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called

1962 on a reference where the target of the reference has become unavailable for some reason, the

1963 SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of

1964 the reference is changed, the reference MAY continue to work, depending on the runtime and the

1965 type of change that was made. [JCA90029] If it doesn't work, the exception thrown will depend on

1966 the runtime and the cause of the failure.

1967 A ServiceReference that has been obtained from a reference by ComponentContext.cast()

1968 corresponds to the reference that is passed as a parameter to cast(). If the reference is

1969 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue

1970 to work as if the reference target was not changed. [JCA90030] If the target of a ServiceReference

1971 has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an

1972 operation is invoked on the ServiceReference. [JCA90031] If the target of a ServiceReference has

1973 become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an

1974 operation is invoked on the ServiceReference. [JCA90032] If the target service of a

1975 ServiceReference is changed, the reference MAY continue to work, depending on the runtime and

1976 the type of change that was made. [JCA90033] If it doesn't work, the exception thrown will

1977 depend on the runtime and the cause of the failure.

1978 A reference or ServiceReference accessed through the component context by calling getService()

1979 or getServiceReference() MUST correspond to the current configuration of the domain. This applies

1980 whether or not reinjection has taken place. [JCA90034] If the target of a reference or

1981 ServiceReference accessed through the component context by calling getService() or

1982 getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a

1983 reference to the undeployed or unavailable service, and attempts to call business methods

1984 SHOULD throw an InvalidServiceException or a ServiceUnavailableException. [JCA90035] If the

1985 target service of a reference or ServiceReference accessed through the component context by

1986 calling getService() or getServiceReference() has changed, the returned value SHOULD be a

1987 reference to the changed service. [JCA90036]

1988 The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This

1989 means that in the cases where reference reinjection is not allowed, the array or Collection for a

1990 reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes

1991 occur to the reference wiring or to the targets of the wiring. [JCA90037] In cases where the

1992 contents of a reference array or collection change when the wiring changes or the targets change,

1993 then for references that use setter injection, the setter method MUST be called by the SCA

1994 runtime for any change to the contents. [JCA90038] A reinjected array or Collection for a

1995 reference MUST NOT be the same array or Collection object previously injected to the component.

1996 [JCA90039]

1997

Change event	Effect on		
	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	continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.

	continues to work as if the reference target was not changed.		
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.
<p>* Other conditions:</p> <p>The component cannot be STATELESS scoped.</p> <p>The reference has to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed.</p> <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

1998

1999 9.18 @Remotable

2000 The following Java code defines the **@Remotable** annotation:

2001

```
2002 package org.oasisopen.sca.annotation;
```

2003

```
2004 import static java.lang.annotation.ElementType.TYPE;
```

```
2005 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
2006 import java.lang.annotation.Retention;
```

```
2007 import java.lang.annotation.Target;
```

2008

2009

```
2010 @Target (TYPE)
```

```
2011 @Retention(RUNTIME)
```

```
2012 public @interface Remotable {
```

2013

2014

2015

```
}
```

2016 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable
 2017 service can be published externally as a service and MUST be translatable into a WSDL portType.
 2018 [JCA90040]

2019 The @Remotable annotation has no attributes.

2020 The following snippet shows the Java interface for a remotable service with its @Remotable
 2021 annotation.

```

2022 package services.hello;
2023
2024 import org.oasisopen.sca.annotation.*;
2025
2026 @Remotable
2027 public interface HelloService {
2028     String hello(String message);
2029 }
2030
2031

```

2032 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
2033 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

2034 Complex data types exchanged via remotable service interfaces need to be compatible with the
2035 marshalling technology used by the service binding. For example, if the service is going to be
2036 exposed using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types
2037 or they can be Service Data Objects (SDOs) [SDO].

2038 Independent of whether the remotable service is called from outside of the composite that
2039 contains it or from another component in the same composite, the data exchange semantics are
2040 **by-value**.

2041 Implementations of remotable services can modify input data during or after an invocation and
2042 can modify return data after the invocation. If a remotable service is called locally or remotely, the
2043 SCA container is responsible for making sure that no modification of input data or post-invocation
2044 modifications to return data are seen by the caller.

2045 The following snippet shows a remotable Java service interface.

```

2046
2047 package services.hello;
2048
2049 import org.oasisopen.sca.annotation.*;
2050
2051 @Remotable
2052 public interface HelloService {
2053     String hello(String message);
2054 }
2055
2056 package services.hello;
2057
2058 import org.oasisopen.sca.annotation.*;
2059
2060 @Service(HelloService.class)
2061 public class HelloServiceImpl implements HelloService {
2062     public String hello(String message) {
2063         ...
2064     }
2065 }
2066
2067

```

2068 9.19 @Requires

2069 The following Java code defines the **@Requires** annotation:

```

2070 package org.oasisopen.sca.annotation;
2071
2072 import static java.lang.annotation.ElementType.FIELD;
2073

```

```

2074 import static java.lang.annotation.ElementType.METHOD;
2075 import static java.lang.annotation.ElementType.PARAMETER;
2076 import static java.lang.annotation.ElementType.TYPE;
2077 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2078
2079 import java.lang.annotation.Inherited;
2080 import java.lang.annotation.Retention;
2081 import java.lang.annotation.Target;
2082
2083 @Inherited
2084 @Retention(RUNTIME)
2085 @Target({TYPE, METHOD, FIELD, PARAMETER})
2086 public @interface Requires {
2087     /**
2088      * Returns the attached intents.
2089      *
2090      * @return the attached intents
2091      */
2092     String[] value() default "";
2093 }
2094

```

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

See the [section "General Intent Annotations"](#) for details and samples.

2098 9.20 @Scope

2099 The following Java code defines the **@Scope** annotation:

```

2100 package org.oasisopen.sca.annotation;
2101
2102 import static java.lang.annotation.ElementType.TYPE;
2103 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2104 import java.lang.annotation.Retention;
2105 import java.lang.annotation.Target;
2106
2107 @Target(TYPE)
2108 @Retention(RUNTIME)
2109 public @interface Scope {
2110
2111     String value() default "STATELESS";
2112 }

```

2113 **The @Scope annotation MUST only be used on a service's implementation class. It is an error to**
2114 **use this annotation on an interface. [JCA90041]**

2115 The @Scope annotation has the following attribute:

- 2116 • **value** – the name of the scope.
- 2117 SCA defines the following scope names, but others can be defined by particular Java-
- 2118 based implementation types:
- 2119 STATELESS
- 2120 COMPOSITE
- 2121 For 'STATELESS' implementations, a different implementation instance can be used to
- 2122 service each request. Implementation instances can be newly created or be drawn from a
- 2123 pool of instances.

2124 The default value is STATELESS.

2125 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```

2126 package services.hello;

```

```

2127
2128 import org.oasisopen.sca.annotation.*;
2129
2130 @Service(HelloService.class)
2131 @Scope("COMPOSITE")
2132 public class HelloServiceImpl implements HelloService {
2133     public String hello(String message) {
2134         ...
2135     }
2136 }
2137
2138

```

2139 9.21 @Service

2140 The following Java code defines the **@Service** annotation:

```

2141 package org.oasisopen.sca.annotation;
2142
2143 import static java.lang.annotation.ElementType.TYPE;
2144 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2145 import java.lang.annotation.Retention;
2146 import java.lang.annotation.Target;
2147
2148 @Target(TYPE)
2149 @Retention(RUNTIME)
2150 public @interface Service {
2151
2152     Class<?>[] interfaces() default {};
2153     Class<?> value() default Void.class;
2154 }
2155

```

2156 The @Service annotation is used on a component implementation class to specify the SCA services
2157 offered by the implementation. **An implementation class need not be declared as implementing all**
2158 **of the interfaces implied by the services declared in its @Service annotation, but all methods of all**
2159 **the declared service interfaces MUST be present.** [JCA90042] A class used as the implementation
2160 of a service is not required to have a @Service annotation. If a class has no @Service annotation,
2161 then the rules determining which services are offered and what interfaces those services have are
2162 determined by the specific implementation type.

2163 The @Service annotation has the following attributes:

- 2164 • **interfaces (1..1)** – The value is an array of interface or class objects that are exposed as
2165 services by this component implementation.
- 2166 • **value** – A shortcut for the case when the class provides only a single service interface -
2167 contains a single interface or class object that is exposed as a service by this component
2168 implementation.

2169 **A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.**
2170 [JCA90043]

2171
2172 **A @Service annotation with no attributes MUST be ignored, it is the same as not having the**
2173 **annotation there at all.** [JCA90044]

2174 The **service names** of the defined services default to the names of the interfaces or class, without
2175 the package name.

2176 **A component implementation MUST NOT have two services with the same Java simple name.**
2177 **[JCA90045]** If a Java implementation needs to realize two services with the same Java simple
2178 name then this can be achieved through subclassing of the interface.

2179 The following snippet shows an implementation of the HelloService marked with the @Service
2180 annotation.

```
2181 package services.hello;  
2182  
2183 import org.oasisopen.sca.annotation.Service;  
2184  
2185 @Service(HelloService.class)  
2186 public class HelloServiceImpl implements HelloService {  
2187     public void hello(String name) {  
2188         System.out.println("Hello " + name);  
2189     }  
2190 }  
2191  
2192
```


2193 10 WSDL to Java and Java to WSDL

2194 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
2195 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java
2196 interfaces from WSDL portTypes and vice versa.

2197 For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java
2198 interface as if it had a @WebService annotation on the class, even if it doesn't. [JCA100001] The
2199 SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for
2200 the @javax.jws.OneWay annotation. [JCA100002] For the WSDL-to-Java mapping, the SCA
2201 runtime MUST take the generated @WebService annotation to imply that the Java interface is
2202 @Remotable. [JCA100003]

2203 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
2204 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
2205 from Java types to XML schema types. [JCA100004] SCA runtimes MAY support the SDO 2.1
2206 mapping from Java types to XML schema types. [JCA100005] Having a choice of binding
2207 technologies is allowed, as noted in the first paragraph of section 5 of the JSR 181 (version 2)
2208 specification, which is referenced by the JAX-WS specification.

2209 The JAX-WS mappings are applied with the following restrictions:

- 2210 • No support for holders

2211

2212 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
2213 model is used.

2214 10.1 JAX-WS Client Asynchronous API for a Synchronous Service

2215 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
2216 application with a means of invoking that service asynchronously, so that the client can invoke a service
2217 operation and proceed to do other work without waiting for the service operation to complete its
2218 processing. The client application can retrieve the results of the service either through a polling
2219 mechanism or via a callback method which is invoked when the operation completes.

2220 For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the
2221 additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006] For
2222 SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional
2223 client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100007] If the additional
2224 client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface
2225 which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these
2226 methods in the SCA reference interface in the component type of the implementation. [JCA100008]
2227

2228 The additional client-side asynchronous polling and callback methods defined by JAX-WS are recognized
2229 in a Java interface as follows:

2230 For each method M in the interface, if another method P in the interface has

- 2231 a. a method name that is M's method name with the characters "Async" appended, and
- 2232 b. the same parameter signature as M, and
- 2233 c. a return type of Response<R> where R is the return type of M

2234 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

2235 For each method M in the interface, if another method C in the interface has

- 2236 a. a method name that is M's method name with the characters "Async" appended, and
- 2237 b. a parameter signature that is M's parameter signature with an additional final parameter of type
2238 AsyncHandler<R> where R is the return type of M, and

2239 c. a return type of Future<?>

2240 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

2241 As an example, an interface can be defined in WSDL as follows:

```
2242 <!-- WSDL extract -->
2243 <message name="getPrice">
2244   <part name="ticker" type="xsd:string"/>
2245 </message>
2246
2247 <message name="getPriceResponse">
2248   <part name="price" type="xsd:float"/>
2249 </message>
2250
2251 <portType name="StockQuote">
2252   <operation name="getPrice">
2253     <input message="tns:getPrice"/>
2254     <output message="tns:getPriceResponse"/>
2255   </operation>
2256 </portType>
```

2257

2258 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2259 // asynchronous mapping
2260 @WebService
2261 public interface StockQuote {
2262   float getPrice(String ticker);
2263   Response<Float> getPriceAsync(String ticker);
2264   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2265 }
```

2266

2267 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2268 // synchronous mapping
2269 @WebService
2270 public interface StockQuote {
2271   float getPrice(String ticker);
2272 }
```

2273

2274 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] In
2275 the above example, if the client implementation uses the asynchronous form of the interface, the
2276 two additional getPriceAsync() methods can be used for polling and callbacks as defined by the
2277 JAX-WS specification.

2278

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

```
2279 <?xml version="1.0" encoding="UTF-8"?>
2280 <!-- (c) Copyright SCA Collaboration 2006 -->
2281 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2282         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2283         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2284         elementFormDefault="qualified">
2285
2286     <include schemaLocation="sca-core.xsd"/>
2287
2288     <element name="interface.java" type="sca:JavaInterface"
2289             substitutionGroup="sca:interface"/>
2290     <complexType name="JavaInterface">
2291         <complexContent>
2292             <extension base="sca:Interface">
2293                 <sequence>
2294                     <any namespace="##other" processContents="lax"
2295                         minOccurs="0" maxOccurs="unbounded"/>
2296                 </sequence>
2297                 <attribute name="interface" type="NCName" use="required"/>
2298                 <attribute name="callbackInterface" type="NCName"
2299                     use="optional"/>
2300                 <anyAttribute namespace="##any" processContents="lax"/>
2301             </extension>
2302         </complexContent>
2303     </complexType>
2304 </schema>
2305
```

2306 **B. Java Classes and Interfaces**

2307

2308 **B.1 SCAClient Classes and Interfaces**

2309

2310 **B.1.1 SCAClient Interface**

2311

```
2312 /*  
2313 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
2314 * OASIS trademark, IPR and other policies apply.  
2315 */  
2316 package org.oasisopen.sca.client;  
2317  
2318 import java.net.URI;  
2319  
2320 import org.oasisopen.sca.NoSuchDomainException;  
2321 import org.oasisopen.sca.NoSuchServiceException;  
2322  
2323 /**  
2324 * Client side interface that can be used to lookup SCA Services within  
2325 * a SCA domain.  
2326 * <p>  
2327 * The SCAClientFactory is used to obtain an implementation instance of  
2328 * the SCAClient.  
2329 *  
2330 * @see SCAClientFactory  
2331 * @author OASIS Open  
2332 */  
2333 public interface SCAClient {  
2334  
2335 /**  
2336 * Returns a reference proxy that implements the business interface <T>  
2337 * of a service in a domain  
2338 *  
2339 * @param serviceURI the relative URI of the target service. Takes the  
2340 * form componentName/serviceName.  
2341 * Can also take the extended form componentName/serviceName/bindingName  
2342 * to use a specific binding of the target service  
2343 *  
2344 * @param domainURI the URI of an SCA Domain.  
2345 * @param interfaze The business interface class of the service in the  
2346 * domain  
2347 * @param <T> The business interface class of the service in the domain  
2348 *  
2349 * @return a proxy to the target service, in the specified SCA Domain  
2350 * that implements the business interface <B>.  
2351 * @throws NoSuchServiceException Service requested was not found  
2352 * @throws NoSuchDomainException Domain requested was not found  
2353 */  
2354 <T> T getService(Class<T> interfaze, String serviceURI, URI domainURI)  
2355 throws NoSuchServiceException, NoSuchDomainException;
```

2356
2357
2358

```
}
```

2359 **B.1.2 SCAClientFactory Class**

2360

2361

```
/*
```

2362

```
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
```

2363

```
 * OASIS trademark, IPR and other policies apply.
```

2364

```
 */
```

2365

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

2366

```
 import java.util.Properties;
```

2368

```
 import org.oasisopen.sca.client.spi.SCAClientFactoryFinder;
```

2370

2371

```
 /**
```

2372

```
 * The SCAClientFactory can be used by non-SCA managed code to
```

2373

```
 * lookup services that exist in a SCADomain.
```

2374

```
 *
```

2375

```
 * @see SCAClientFactoryFinder
```

2376

```
 * @see SCAClient
```

2377

```
 *
```

2378

```
 * @author OASIS Open
```

2379

```
 */
```

2380

```
 public abstract class SCAClientFactory {
```

2381

2382

```
     /**
```

2383

```
      * The default implementation of the SCAClientFactory.
```

2384

```
      * A Vendor may use reflection to inject a default
```

2385

```
      * SCAClientFactory instance that will be used in the
```

2386

```
      * newInstance() methods rather than using the
```

2387

```
      * SCAClientFactoryFinder.
```

2388

```
      */
```

2389

```
      private static SCAClientFactory defaultFactory;
```

2390

2391

```
     /**
```

2392

```
      * Creates a new instance of the SCAClient that can be
```

2393

```
      * used to lookup SCA Services.
```

2394

```
      *
```

2395

```
      * @return A new SCAClient
```

2396

```
      */
```

2397

```
      public static SCAClient newInstance() {
```

2398

```
          return newInstance(null, null);
```

2399

```
      }
```

2400

2401

```
     /**
```

2402

```
      * Creates a new instance of the SCAClient that can be
```

2403

```
      * used to lookup SCA Services.
```

2404

```
      *
```

2405

```
      * @param properties Properties that may be used when
```

2406

```
      * creating a new instance of the SCAClient
```

2407

```
      * @return A new SCAClient instance
```

2408

```
      */
```

2409

```
      public static SCAClient newInstance(Properties properties) {
```

2410

```
          return newInstance(properties, null);
```

```

2411     }
2412
2413     /**
2414     * Creates a new instance of the SCAClient that can be
2415     * used to lookup SCA Services.
2416     *
2417     * @param classLoader ClassLoader that may be used when
2418     * creating a new instance of the SCAClient
2419     * @return A new SCAClient instance
2420     */
2421     public static SCAClient newInstance(ClassLoader classLoader) {
2422         return newInstance(null, classLoader);
2423     }
2424
2425     /**
2426     * Creates a new instance of the SCAClient that can be
2427     * used to lookup SCA Services.
2428     *
2429     * @param properties Properties that may be used when
2430     * creating a new instance of the SCAClient
2431     * @param classLoader ClassLoader that may be used when
2432     * creating a new instance of the SCAClient
2433     * @return A new SCAClient instance
2434     */
2435     public static SCAClient newInstance(Properties properties,
2436                                       ClassLoader classLoader) {
2437         final SCAClientFactory factory;
2438         if(defaultFactory == null) {
2439             factory = SCAClientFactoryFinder.find(properties,
2440                                                  classLoader);
2441         } else {
2442             factory = defaultFactory;
2443         }
2444         return factory.createSCAClient();
2445     }
2446
2447     /**
2448     * This method is invoked to create a new SCAClient instance.
2449     *
2450     * @return A new SCAClient instance
2451     */
2452     protected abstract SCAClient createSCAClient();
2453 }
2454
2455

```

2456 **B.1.3 SCAFactoryFinder class**

2457 SCA provides a reference implementation of the SCAClientFactory class. It discovers a vendor's
2458 SCAClientFactory implementation by referring to the following information:

- 2459 • The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the
2460 newInstance() method call if specified
- 2461 • The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 2462 • The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

2463

2464 Since this is a reference implementation, vendors are free to replace the SCAClientFactoryFinder class
2465 with an alternative implementation that provides the lookup mechanisms required for their SCA Runtime.

```
2466
2467 /*
2468  * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
2469  * OASIS trademark, IPR and other policies apply.
2470 */
2471 package org.oasisopen.sca.client.spi;
2472
2473 import java.io.BufferedReader;
2474 import java.io.Closeable;
2475 import java.io.IOException;
2476 import java.io.InputStream;
2477 import java.io.InputStreamReader;
2478 import java.net.URL;
2479 import java.util.Properties;
2480
2481 import org.oasisopen.sca.SCARuntimeException;
2482 import org.oasisopen.sca.client.SCAClientFactory;
2483
2484 /**
2485  * This is the interface that a SCA Runtime Provider can
2486  * implement for constructing implementation specific instances of
2487  * objects offering the SCAClient interface.
2488  *
2489  * @see SCAClientFactory
2490  *
2491  * @author OASIS Open
2492 */
2493 public class SCAClientFactoryFinder {
2494
2495     /**
2496      * The name of the System Property used to determine the SPI
2497      * implementation to use for the SCAClientFactory.
2498      */
2499     private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
2500         SCAClientFactory.class.getName();
2501
2502     /**
2503      * The name of the file loaded from the ClassPath to determine
2504      * the SPI implementation to use for the SCAClientFactory.
2505      */
2506     private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
2507         = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
2508
2509     /**
2510      * Private Constructor.
2511      */
2512     private SCAClientFactoryFinder() {
2513     }
2514
2515     /**
2516      * Creates an instance of the SCAClientFactorySPI implementation.
2517      * This discovers the SCAClientFactorySPI Implementation and instantiates
2518      * the provider's implementation.
2519      *
2520      * @param properties Properties that may be used when creating a new
```

```

2521     * instance of the SCAClient
2522     * @param classLoader ClassLoader that may be used when creating a new
2523     * instance of the SCAClient
2524     * @return new instance of the SCAClientFactorySPI
2525     * @throws SCARuntimeException Failed to create SCAClientFactorySPI
2526     * Implementation.
2527     */
2528     public static SCAClientFactory find(Properties properties,
2529                                         ClassLoader classLoader)
2530     {
2531         if (classLoader == null) {
2532             classLoader = getThreadContextClassLoader();
2533         }
2534         final String factoryImplClassName =
2535             discoverProviderFactoryImplClass(properties, classLoader);
2536         final Class<? extends SCAClientFactory> factoryImplClass
2537             = loadProviderFactoryClass(factoryImplClassName, classLoader);
2538         final SCAClientFactory factory =
2539             instantiateSCAClientFactoryClass(factoryImplClass);
2540         return factory;
2541     }
2542
2543     /**
2544     * Gets the Context ClassLoader for the current Thread.
2545     *
2546     * @return The Context ClassLoader for the current Thread.
2547     */
2548     private static ClassLoader getThreadContextClassLoader() {
2549         final ClassLoader threadClassLoader =
2550             Thread.currentThread().getContextClassLoader();
2551         return threadClassLoader;
2552     }
2553
2554     /**
2555     * Attempts to discover the class name for the SCAClientFactorySPI
2556     * implementation from the specified Properties, the System Properties
2557     * or the specified ClassLoader.
2558     *
2559     * @return The class name of the SCAClientFactorySPI implementation
2560     * @throw SCARuntimeException Failed to find implementation for
2561     * SCAClientFactorySPI.
2562     */
2563     private static String
2564         discoverProviderFactoryImplClass(Properties properties,
2565                                         ClassLoader classLoader)
2566         throws SCARuntimeException {
2567         String providerClassName =
2568             checkPropertiesForSPIClassName(properties);
2569         if (providerClassName != null) {
2570             return providerClassName;
2571         }
2572
2573         providerClassName =
2574             checkPropertiesForSPIClassName(System.getProperties());
2575         if (providerClassName != null) {
2576             return providerClassName;
2577         }
2578     }

```



```

2579     providerClassName = checkMETA-INFServicesForSIPClassName(classLoader);
2580     if (providerClassName == null) {
2581         throw new SCARuntimeException(
2582             "Failed to find implementation for SCAClientFactory");
2583     }
2584
2585     return providerClassName;
2586 }
2587
2588 /**
2589  * Attempts to find the class name for the SCAClientFactorySPI
2590  * implementation from the specified Properties.
2591  *
2592  * @return The class name for the SCAClientFactorySPI implementation
2593  * or <code>null</code> if not found.
2594  */
2595 private static String
2596     checkPropertiesForSPIClassName(Properties properties) {
2597     if (properties == null) {
2598         return null;
2599     }
2600
2601     final String providerClassName =
2602         properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
2603     if (providerClassName != null && providerClassName.length() > 0) {
2604         return providerClassName;
2605     }
2606
2607     return null;
2608 }
2609
2610 /**
2611  * Attempts to find the class name for the SCAClientFactorySPI
2612  * implementation from the META-INF/services directory
2613  *
2614  * @return The class name for the SCAClientFactorySPI implementation or
2615  * <code>null</code> if not found.
2616  */
2617 private static String checkMETA-INFServicesForSIPClassName(ClassLoader cl)
2618 {
2619     final URL url =
2620         cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
2621     if (url == null) {
2622         return null;
2623     }
2624
2625     InputStream in = null;
2626     try {
2627         in = url.openStream();
2628         BufferedReader reader = null;
2629         try {
2630             reader =
2631                 new BufferedReader(new InputStreamReader(in, "UTF-8"));
2632
2633             String line;
2634             while ((line = readNextLine(reader)) != null) {
2635                 if (!line.startsWith("#") && line.length() > 0) {
2636                     return line;

```

```

2637     }
2638 }
2639
2640     return null;
2641 } finally {
2642     closeStream(reader);
2643 }
2644 } catch (IOException ex) {
2645     throw new SCARuntimeException(
2646         "Failed to discover SCAClientFactory provider", ex);
2647 } finally {
2648     closeStream(in);
2649 }
2650 }
2651
2652 /**
2653  * Reads the next line from the reader and returns the trimmed version
2654  * of that line
2655  *
2656  * @param reader The reader from which to read the next line
2657  * @return The trimmed next line or <code>null</code> if the end of the
2658  * stream has been reached
2659  * @throws IOException I/O error occurred while reading from Reader
2660  */
2661 private static String readNextLine(BufferedReader reader)
2662     throws IOException {
2663
2664     String line = reader.readLine();
2665     if (line != null) {
2666         line = line.trim();
2667     }
2668     return line;
2669 }
2670
2671 /**
2672  * Loads the specified SCAClientFactory Implementation class.
2673  *
2674  * @param factoryImplClassName The name of the SCAClientFactory
2675  * Implementation class to load
2676  * @return The specified SCAClientFactory Implementation class
2677  * @throws SCARuntimeException Failed to load the SCAClientFactory
2678  * Implementation class
2679  */
2680 private static Class<? extends SCAClientFactory>
2681     loadProviderFactoryClass(String factoryImplClassName,
2682         ClassLoader classLoader)
2683     throws SCARuntimeException {
2684
2685     try {
2686         final Class<?> providerClass =
2687             classLoader.loadClass(factoryImplClassName);
2688         final Class<? extends SCAClientFactory> providerFactoryClass =
2689             providerClass.asSubclass(SCAClientFactory.class);
2690         return providerFactoryClass;
2691     } catch (ClassNotFoundException ex) {
2692         throw new SCARuntimeException(
2693             "Failed to load SCAClientFactory implementation class "
2694             + factoryImplClassName, ex);

```

```

2695     } catch (ClassCastException ex) {
2696         throw new SCARuntimeException(
2697             "Loaded SCAClientFactory implementation class "
2698             + factoryImplClassName
2699             + " is not a subclass of "
2700             + SCAClientFactory.class.getName() , ex);
2701     }
2702 }
2703
2704 /**
2705  * Instantiate an instance of the specified SCAClientFactorySPI
2706  * Implementation class.
2707  *
2708  * @param factoryImplClass The SCAClientFactorySPI Implementation
2709  * class to instantiate.
2710  * @return An instance of the SCAClientFactorySPI Implementation class
2711  * @throws SCARuntimeException Failed to instantiate the specified
2712  * specified SCAClientFactorySPI Implementation class
2713  */
2714 private static SCAClientFactory
2715     instantiateSCAClientFactoryClass(
2716         Class<? extends SCAClientFactory> factoryImplClass)
2717     throws SCARuntimeException {
2718
2719     try {
2720         final SCAClientFactory provider = factoryImplClass.newInstance();
2721         return provider;
2722     } catch (Throwable ex) {
2723         throw new SCARuntimeException(
2724             "Failed to instantiate SCAClientFactory implementation class "
2725             + factoryImplClass, ex);
2726     }
2727 }
2728
2729 /**
2730  * Utility method for closing Closeable Object.
2731  *
2732  * @param closeable The Object to close.
2733  */
2734 private static void closeStream(Closeable closeable) {
2735     if (closeable != null) {
2736         try{
2737             closeable.close();
2738         } catch (IOException ex) {
2739             throw new SCARuntimeException("Failed to close stream", ex);
2740         }
2741     }
2742 }
2743 }
2744

```

2745 **B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?**

2746 The SCAClient classes and interfaces are designed so that vendors can provide their own
2747 implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor
2748 needs to consider in relation to the SCAClient classes and interfaces.

- 2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
- Implement their SCAClientFactory and SCAClient implementation classes
Vendors need to provide an implementation of SCAClient that is capable of looking up Services in their SCA Runtime.
Vendors need to subclass SCAClientFactory and implement the createSCAClient() method so that it creates an instance of their SCAClient implementation.
 - Configure the Vendor Implementation classes so they are used
Vendors have several options:
 - Option 1: Set System Property to point to the Vendor's implementation
Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their implementation class and use the reference implementation of SCAClientFactoryFinder
 - Option 2: Provide a META-INF/services file
Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points to their implementation class and use the reference implementation of SCAClientFactoryFinder
 - Option 3: Inject a vendor implementation instance into SCAClientFactory
Vendors inject an instance into the defaultFactory field of SCAClientFactory. The SCAClientFactoryFinder is not used in this scenario.
 - Option 4: Provide a Vendor specific implementation of SCAClientFactoryFinder
Vendors write a new implementation of SCAClientFactoryFinder and replace the reference implementation that is provided by SCA.

2780

C. Conformance Items

2781 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2782 specification.

2783

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]	For a composite scope implementation instance, the SCA runtime MUST ensure that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.
[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.
[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.
[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.
[JCA80001]	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`.
- [JCA90012] the `@Property` annotation MUST be used in order to inject a property onto a non-public field.
- [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.
- [JCA90017] the @Reference annotation MUST be used in order to inject a reference onto a non-public field.
- [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 MAY continue to work, depending on the runtime and the type of change that was made.
- [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 MAY continue to work, depending on the runtime and the type of change that was made.
- [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] The @Remotable annotation is used to specify a Java service interface as remotable. A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.
- [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 with no attributes MUST be ignored, it is the same as not having the annotation there at all.
- [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 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.
- [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.

2784

2785

D. Acknowledgements

2786 The following individuals have participated in the creation of this specification and are gratefully
2787 acknowledged:

2788 **Participants:**

2789 [Participant Name, Affiliation | Individual Member]

2790 [Participant Name, Affiliation | Individual Member]

2791

2793

F. Revision History

2794 [optional; should not be included in OASIS Standards]

2795

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 * Note that issue 33 was applied, but not noted, in a previous version * Replaced the osoa.org NS with the oasis-open.org NS
WD05	2008-10-03	Anish Karmarkar	* Fixed the resolution of issue 37 but re-adding the sentence: "However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section * resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references > 1 * minor ed changes
cd01-rev1	2008-12-11	Anish Karmarkar	* Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup. All changes accepted.

			All comments removed.
cd02-rev1	2009-02-03	Mike Edwards	Issues 25+95 Issue 120
cd02-rev2	2009-02-08	Mike Edwards	Merge annotation definitions contained in section 10 into section 8 Move remaining parts of section 10 to section 7. Accept all changes.
cd02-rev3	2009-03-16	Mike Edwards	RFC2119 work and formal marking of all normative statements - all sections. Completion of Appendix B (list of all normative statements) Accept all changes

2796