



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 02

08 February 2009

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev2.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev2.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev2.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 Combellack, | Avaya |

Editor(s):

| | |
|------------------|--------|
| Ron Barack, | SAP |
| David Booz, | IBM |
| Mark Combellack, | 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 chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

Status:

This document was last revised or approved by the OASIS Service Component Architecture / J (SCA-J) TC on the above date. The level of approval is also listed above. Check the "Latest Version" or "Latest Approved Version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at <http://www.oasis-open.org/committees/sca-j/>.

For information on whether any patents have been disclosed that might be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (<http://www.oasis-open.org/committees/sca-j/ipr.php>).

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

Notices

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

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

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

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

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

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

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

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

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

Table of Contents

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

| | | |
|--------|---|----|
| 8.1 | Component Context..... | 29 |
| 8.2 | Request Context | 30 |
| 8.3 | ServiceReference | 31 |
| 8.4 | ServiceRuntimeException..... | 31 |
| 8.5 | ServiceUnavailableException | 32 |
| 8.6 | InvalidServiceException..... | 32 |
| 8.7 | Constants Interface..... | 32 |
| 9 | Java Annotations | 33 |
| 9.1 | @AllowsPassByReference | 33 |
| 9.2 | @Authentication | 34 |
| 9.3 | @Callback | 34 |
| 9.4 | @ComponentName | 35 |
| 9.5 | @Confidentiality..... | 36 |
| 9.6 | @Constructor..... | 37 |
| 9.7 | @Context..... | 37 |
| 9.8 | @Destroy..... | 38 |
| 9.9 | @EagerInit..... | 39 |
| 9.10 | @Init..... | 39 |
| 9.11 | @Integrity | 40 |
| 9.12 | @Intent | 40 |
| 9.13 | @OneWay | 41 |
| 9.14 | @PolicySet | 42 |
| 9.15 | @Property..... | 42 |
| 9.16 | @Qualifier | 44 |
| 9.17 | @Reference..... | 44 |
| 9.17.1 | Reinjection..... | 47 |
| 9.18 | @Remotable | 48 |
| 9.19 | @Requires..... | 49 |
| 9.20 | @Scope | 50 |
| 9.21 | @Service | 51 |
| 10 | WSDL to Java and Java to WSDL | 53 |
| 10.1 | JAX-WS Client Asynchronous API for a Synchronous Service..... | 53 |
| A. | XML Schema: sca-interface-java.xsd..... | 55 |
| B. | Conformance Items | 56 |
| C. | Acknowledgements | 57 |
| D. | Non-Normative Text | 58 |
| E. | Revision History..... | 59 |

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 chose 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, client and implementation model defined in this specification are designed to be used by other SCA Java-related specifications in either a partial or complete fashion.

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

1.1 Terminology

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

1.2 Normative References

- | | |
|------------|---|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , 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/wsdl20/ |
| [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>

2 Implementation Metadata

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

2.1 Service Metadata

2.1.1 @Service

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

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

2.1.2 Java Semantics of a Remotable Service

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

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

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

2.1.3 Java Semantics of a Local Service

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

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

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

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

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

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

95 2.1.4 @Reference

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

99 2.1.5 @Property

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

103 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

110 This document defines two scopes:

- 111 • STATELESS
- 112 • COMPOSITE

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

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

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

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

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

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

```
125     @Init  
126     public void start() {  
127         ...  
128     }  
129  
130     @Destroy  
131     public void stop() {  
132         ...  
133     }  
134  
135
```

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

138 2.2.1 Stateless scope

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

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

147 **2.2.2 Composite scope**

148 All service requests are dispatched to the same implementation instance for the lifetime of the
149 containing composite. 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 A composite scoped implementation can also specify eager initialization using the **@EagerInit**
152 annotation. When marked for eager initialization, the composite scoped instance is created when
153 its containing component is started. If a method is marked with the @Init annotation, it is called
154 when the instance is created.

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 it MUST NOT perform any synchronization.

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
163 optionally identifies a callback interface, where the first Java interface represents the forward
164 (service) call interface and the second interface represents the interface used to call back from the
165 service to the client.

166
167 The following is the pseudo-schema for the interface.java element

```
168  
169 <interface.java interface="NCName" callbackInterface="NCName"? />  
170
```

171 The interface.java element has the following attributes:

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

177
178 The following snippet shows an example of the Java interface element:

```
179  
180 <interface.java interface="services.stockquote.StockQuoteService"  
181     callbackInterface="services.stockquote.StockQuoteServiceCallback" />  
182
```

183 Here, the Java interface is defined in the Java class file
184 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
185 contribution in which the interface exists. Similarly, the callback interface is defined in the Java
186 class file ./services/stockquote/StockQuoteServiceCallback.class.

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

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

199 3.2 @Remotable

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

204 3.3 @Callback

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

209 4 Client API

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

212 4.1 Accessing Services from an SCA Component

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

217 4.1.1 Using the Component Context API

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

221 4.2 Accessing Services from non-SCA component implementations

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

224 4.2.1 ComponentContext

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

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

```
229  
230 ComponentContext context = // obtained via host environment-specific means  
231 HelloService helloService =  
232     context.getService(HelloService.class, "HelloService");  
233 String result = helloService.hello("Hello World!");
```

234 5 Error Handling

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

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

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

241 6 Asynchronous Programming

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

- 249 • support for non-blocking method calls
- 250 • callbacks

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

252 6.1 @OneWay

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

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

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

263 6.2 Callbacks

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

- 268 • an interface for the provided service
- 269 • a callback interface that must be provided by the client

270 Callbacks can be used for both remotable and local services. Either both interfaces of a
271 bidirectional service must be remotable, or both must be local. It is illegal to mix the two.

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

276 6.2.1 Using Callbacks

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

282 The following example shows a scenario in which bidirectional interfaces and callbacks could be
283 used. A client requests a quotation from a supplier. To process the enquiry and return the
284 quotation, some suppliers might need additional information from the client. The client does not

285 know which additional items of information will be needed by different suppliers. This interaction
286 can be modeled as a bidirectional interface with callback requests to obtain the additional
287 information.

```
288 package somepackage;
289 import org.osoa.sca.annotation.Callback;
290 import org.osoa.sca.annotation.Remotable;
291 @Remotable
292 @Callback(QuotationCallback.class)
293 public interface Quotation {h
294     double requestQuotation(String productCode, int quantity);
295 }
296
297 @Remotable
298 public interface QuotationCallback {
299     String getState();
300     String getZipCode();
301     String getCreditRating();
302 }
303
```

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

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

```
313 @Callback
314 protected QuotationCallback callback;
315
316 public double requestQuotation(String productCode, int quantity) {
317     double price = getPrice(productCode, quantity);
318     double discount = 0;
319     if (quantity > 1000 && callback.getState().equals("FL")) {
320         discount = 0.05;
321     }
322     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
323         discount += 0.05;
324     }
325     return price * (1-discount);
326 }
327
328
```

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

```
332
333 public class ClientImpl implements ClientService, QuotationCallback {
334
335     private QuotationService myService;
336
337     @Reference
338     public void setMyService(QuotationService service) {
339         myService = service;
340     }

```



```

341
342     public void aClientMethod() {
343         ...
344         double quote = myService.requestQuotation("AB123", 2000);
345         ...
346     }
347
348     public String getState() {
349         return "TX";
350     }
351     public String getZipCode() {
352         return "78746";
353     }
354     public String getCreditRating() {
355         return "AA";
356     }
357 }

```

358
359 In this example the callback is *stateless*, i.e., the callback requests do not need any information
360 relating to the original service request. For a callback that needs information relating to the
361 original service request (a *stateful* callback), this information can be passed to the client by the
362 service provider as parameters on the callback request..

363 6.2.2 Callback Instance Management

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

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

375 6.2.3 Implementing Multiple Bidirectional Interfaces

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

```

382
383 @Callback
384 protected MyService1Callback callback1;
385
386 @Callback
387 protected MyService2Callback callback2;

```

388
389 If a single callback has a type that is compatible with multiple declared callback fields, then all of
390 them will be set.

391 6.2.4 Accessing Callbacks

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

395
396 A reference implementing the callback service interface can be obtained using
397 `ServiceReference.getService()`.

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

```
399 @Callback  
400 protected ServiceReference<MyCallback> callback;  
401  
402 public void someMethod() {  
403     MyCallback myCallback = callback.getCallback();    ...  
404  
405     myCallback.receiveResult(theResult);  
406 }  
407  
408  
409
```

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

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

```
416 public void someMethod() {  
417     MyCallback myCallback =  
418         ComponentContext.getRequestContext().getCallback();  
419  
420     ...  
421  
422     myCallback.receiveResult(theResult);  
423 }  
424  
425
```

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

```
428 @Context  
429 protected RequestContext requestContext;  
430  
431 void receiveResult(Object theResult) {  
432     Object refParams =  
433         requestContext.getServiceReference().getCallbackID();  
434     ...  
435 }  
436
```

437
438 This is necessary if the service implementation has COMPOSITE scope, because callback injection
439 is not performed for composite-scoped implementations.

7 Policy Annotations for Java

440
441 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which
442 influence how implementations, services and references behave at runtime. The policy facilities
443 are described in [the SCA Policy Framework specification \[POLICY\]](#). In particular, the facilities
444 include Intents and Policy Sets, where intents express abstract, high-level policy requirements and
445 policy sets express low-level detailed concrete policies.

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

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

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

463 The SCA Java Common Annotations specification supports using [the Common Annotation for Java
464 Platform specification \(JSR-250\) \[JSR-250\]](#). An implication of adopting the common annotation
465 for Java platform specification is that the SCA Java specification support consistent annotation and
466 Java class inheritance relationships.

467

7.1 General Intent Annotations

468
469 SCA provides the annotation **@Requires** for the attachment of any intent to a Java class, to a
470 Java interface or to elements within classes and interfaces such as methods and fields.

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

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

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

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

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

```
481     public static final String SCA_PREFIX=  
482         "{http://docs.oasis-open.org/ns/opencsa/sca/200712}";  
483     public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
484     public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
485
```

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

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

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

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

494

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

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

```
497     package com.foo;
498
499     import static org.oasisopen.sca.annotation.Confidentiality.*;
500     import static org.oasisopen.sca.annotation.Reference;
501     import static org.oasisopen.sca.annotation.Requires;
502
503     public class Foo {
504         @Requires(CONFIDENTIALITY)
505         @Reference
506         public void setBar(Bar bar) {
507             ...
508         }
509     }
510
```

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

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

528 The formal syntax for the @Requires annotation follows:

```
529     @Requires( "qualifiedIntent" (, "qualifiedIntent")* )
```

530 where

```
531     qualifiedIntent ::= QName(.qualifier)*
```

532

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

534 7.2 Specific Intent Annotations

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

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

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

```
546 @Integrity
```

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

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

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

552 The general form of specific intent annotations is:

```
553 @<Intent>[(qualifiers)]
```

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

```
555 Intent      ::= NCName  
556 qualifiers  ::= "qualifier" (, "qualifier")*  
557 qualifier   ::= NCName(.qualifier)?  
558
```

559 7.2.1 How to Create Specific Intent Annotations

560 SCA identifies annotations that correspond to intents by providing an @Intent annotation which
561 must be used in the definition of an intent annotation.

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

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

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

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

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

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

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

579 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
580 are set for the element to which the confidentiality intent is attached.

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

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

584 7.3 Application of Intent Annotations

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

- 586 • Java class
- 587 • Java interface
- 588 • Method
- 589 • Field
- 590 • Constructor parameter

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

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

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

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

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

603 7.3.1 Inheritance And Annotation

604 The inheritance rules for annotations are consistent with the common annotation specification, JSR
605 250.

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

```
608     package services.hello;  
609     import org.oasisopen.sca.annotation.Remotable;  
610     import org.oasisopen.sca.annotation.Integrity;  
611     import org.oasisopen.sca.annotation.Authentication;  
612  
613     @Integrity("transport")  
614     @Authentication  
615     public class HelloService {  
616         @Integrity  
617         @Authentication("message")  
618         public String hello(String message) {...}  
619  
620         @Integrity  
621         @Authentication("transport")  
622         public String helloThere() {...}  
623     }  
624  
625     package services.hello;  
626     import org.oasisopen.sca.annotation.Remotable;  
627     import org.oasisopen.sca.annotation.Confidentiality;  
628     import org.oasisopen.sca.annotation.Authentication;
```

```

629
630     @Confidentiality("message")
631     public class HelloChildService extends HelloService {
632         @Confidentiality("transport")
633         public String hello(String message) {...}
634         @Authentication
635         String helloWorld() {...}
636     }

```

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

638

639 The effective intent annotation on the helloWorld method is Integrity("transport"),
640 @Authentication, and @Confidentiality("message").

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

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

645 The effective intent annotation on the hello method of the HelloService is @Integrity and
646 @Authentication("message")

647

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

651

```

652     <?xml version="1.0" encoding="ASCII"?>
653     <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
654             name="HelloServiceComposite" >
655         <service name="HelloService" requires="integrity/transport
656             authentication">
657             ...
658         </service>
659         <service name="HelloChildService" requires="integrity/transport
660             authentication confidentiality/message">
661             ...
662         </service>
663         ...
664
665         <component name="HelloServiceComponent">*
666             <implementation.java class="services.hello.HelloService"/>
667             <operation name="hello" requires="integrity
668                 authentication/message"/>
669             <operation name="helloThere"
670                 requires="integrity
671                     authentication/transport"/>
672         </component>
673         <component name="HelloChildServiceComponent">*
674             <implementation.java
675                 class="services.hello.HelloChildService" />
676             <operation name="hello"
677                 requires="confidentiality/transport"/>
678             <operation name="helloThere" requires=" integrity/transport
679                 authentication"/>
680             <operation name="helloWorld" requires="authentication"/>
681         </component>
682

```

```
683         ...
684
685     </composite>
```

686 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

688

689 7.4 Relationship of Declarative And Annotated Intents

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

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

697 7.5 Policy Set Annotations

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

701 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
702 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
703 of two or more policy sets as an array of strings:
704

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

706

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

708 An example of the @PolicySets annotation:

709

```
710     @Reference(name="helloService", required=true)
711     @PolicySets({ MY_NS + "WS_Encryption_Policy",
712                 MY_NS + "WS_Authentication_Policy" })
713     public setHelloService(HelloService service) {
714         . . .
715     }
```

716

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

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

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

- 722 • Java class
- 723 • Java interface
- 724 • Method
- 725 • Field
- 726 • Constructor parameter

727 7.6 Security Policy Annotations

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

730 7.6.1 Security Interaction Policy

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

- 733 • @Integrity
- 734 • @Confidentiality
- 735 • @Authentication

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

- 737 • message
- 738 • transport

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

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

```
744
745 package services.hello;
746 //Interface for HelloService
747 public interface HelloService {
748     String hello(String helloMsg);
749 }
750
751 package services.client;
752 // Interface for ClientService
753 public interface ClientService {
754     public void clientMethod();
755 }
756
757 // Implementation class for ClientService
758 package services.client;
759
760 import services.hello.HelloService;
761 import org.oasisopen.sca.annotation.*;
762
763 @Service(ClientService.class)
764 public class ClientServiceImpl implements ClientService {
765
766     private HelloService helloService;
767
768     @Reference(name="helloService", required=true)
769     @Integrity("message")
770     @Authentication("message")
771     public void setHelloService(HelloService service) {
772         helloService = service;
773     }
774
775     public void clientMethod() {
776         String result = helloService.hello("Hello World!");
```

```
777         ...
778     }
779 }
```

780

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

782 7.6.2 Security Implementation Policy

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

786 • RunAs

787

788 Takes as a parameter a string which is the name of a Security role.

789 eg. @RunAs("Manager")

790 • Code marked with this annotation will execute with the Security permissions of the
791 identified role.

792 • RolesAllowed

793

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

799 • PermitAll

800

801 No parameters. When present, grants access to all roles.

802 • DenyAll

803

804 No parameters. When present, denies access to all roles.

805 • DeclareRoles

806 Takes as a parameter a string or an array of strings which identify one or more role names
807 that form the set of roles used by the implementation.

808 eg. @DeclareRoles({"Manager", "Employee", "Customer"})

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

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

811 7.6.2.1 Annotated Implementation Policy Example

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

813

```
814 package services.account;
```

```
815 @Remotable
```

```
816 public interface AccountService {
```

```
817     AccountReport getAccountReport(String customerID);
```

```
818     float fromUSDollarToCurrency(float value);
```

```
819 }
```

820

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

824

```

825 package services.account;
826 import java.util.List;
827 import commonj.sdo.DataFactory;
828 import org.oasisopen.sca.annotation.Property;
829 import org.oasisopen.sca.annotation.Reference;
830 import org.oasisopen.sca.annotation.RolesAllowed;
831 import org.oasisopen.sca.annotation.RunAs;
832 import org.oasisopen.sca.annotation.PermitAll;
833 import services.accountdata.AccountDataService;
834 import services.accountdata.CheckingAccount;
835 import services.accountdata.SavingsAccount;
836 import services.accountdata.StockAccount;
837 import services.stockquote.StockQuoteService;
838 @RolesAllowed("customers")
839 @RunAs("accountants" )
840 public class AccountServiceImpl implements AccountService {
841
842     @Property
843     protected String currency = "USD";
844
845     @Reference
846     protected AccountDataService accountDataService;
847     @Reference
848     protected StockQuoteService stockQuoteService;
849
850     @RolesAllowed({"customers", "accountants"})
851     public AccountReport getAccountReport(String customerID) {
852
853         DataFactory dataFactory = DataFactory.INSTANCE;
854         AccountReport accountReport =
855             (AccountReport)dataFactory.create(AccountReport.class);
856         List accountSummaries = accountReport.getAccountSummaries();
857
858         CheckingAccount checkingAccount =
859             accountDataService.getCheckingAccount(customerID);
860         AccountSummary checkingAccountSummary =
861             (AccountSummary)dataFactory.create(AccountSummary.class);
862
863         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
864 );
865         checkingAccountSummary.setAccountType("checking");
866         checkingAccountSummary.setBalance(fromUSDollarToCurrency
867             (checkingAccount.getBalance()));
868         accountSummaries.add(checkingAccountSummary);
869
870         SavingsAccount savingsAccount =
871             accountDataService.getSavingsAccount(customerID);
872         AccountSummary savingsAccountSummary =
873             (AccountSummary)dataFactory.create(AccountSummary.class);
874
875         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
876         savingsAccountSummary.setAccountType("savings");
877         savingsAccountSummary.setBalance(fromUSDollarToCurrency
878             (savingsAccount.getBalance()));
879         accountSummaries.add(savingsAccountSummary);
880
881         StockAccount stockAccount =
882         accountDataService.getStockAccount(customerID);

```

```

883     AccountSummary stockAccountSummary =
884         (AccountSummary)dataFactory.create(AccountSummary.class);
885     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
886     stockAccountSummary.setAccountType("stock");
887     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
888         stockAccount.getQuantity();
889     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
890     accountSummaries.add(stockAccountSummary);
891
892     return accountReport;
893 }
894
895 @PermitAll
896 public float fromUSDollarToCurrency(float value) {
897
898     if (currency.equals("USD")) return value;
899     if (currency.equals("EURO")) return value * 0.8f;
900     return 0.0f;
901 }
902 }

```

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

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

- 905 • @RolesAllowed("customers") - indicating that customers have access to the
- 906 implementation as a whole
- 907 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 908 permissions of accountants

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

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

913 8 Java API

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

915 8.1 Component Context

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

```
917
918 package org.oasisopen.sca;
919
920 public interface ComponentContext {
921     String getURI();
922
923     <B> B getService(Class<B> businessInterface, String referenceName);
924
925     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
926                                             String referenceName);
927
928     <B> Collection<B> getServices(Class<B> businessInterface,
929                                String referenceName);
930
931     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
932                                                           businessInterface, String referenceName);
933
934     <B> ServiceReference<B> createSelfReference(Class<B>
935                                                businessInterface);
936
937     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
938                                                String serviceName);
939
940     <B> B getProperty(Class<B> type, String propertyName);
941
942     <B, R extends ServiceReference<B>> R cast(B target)
943         throws IllegalArgumentException;
944
945     RequestContext getRequestContext();
946
947
948 }
```

949

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

951 • **getService(Class businessInterface, String referenceName)** – Returns a proxy for
952 the reference defined by the current component. The getService() method takes as its
953 input arguments the Java type used to represent the target service on the client and the
954 name of the service reference. It returns an object providing access to the service. The
955 returned object implements the Java interface the service is typed with. This method
956 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
957 one.

958 • **getServiceReference(Class businessInterface, String referenceName)** – Returns a
959 ServiceReference defined by the current component. This method MUST throw an
960 IllegalArgumentException if the reference has multiplicity greater than one.

- 961 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
962 typed service proxies for a business interface type and a reference name.
- 963 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
964 list typed service references for a business interface type and a reference name.
- 965 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
966 be used to invoke this component over the designated service.
- 967 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
968 ServiceReference that can be used to invoke this component over the designated service.
969 Service name explicitly declares the service name to invoke
- 970 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
971 property defined by this component.
- 972 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
973 there is no current request or if the context is unavailable. This method MUST return non-
974 null when invoked during the execution of a Java business method for a service operation
975 or callback operation, on the same thread that the SCA runtime provided, and MUST
976 return null in all other cases.
- 977 • **cast(B target)** - Casts a type-safe reference to a ServiceReference

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

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

```
983 private ComponentContext componentContext;
984
985 @Context
986 public void setContext(ComponentContext context) {
987     componentContext = context;
988 }
989
990 public void doSomething() {
991     HelloWorld service =
992     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
993     service.hello("hello");
994 }
995
```

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

999 8.2 Request Context

1000 The following shows the **RequestContext** interface:

```
1001
1002 package org.oasisopen.sca;
1003
1004 import javax.security.auth.Subject;
1005
1006 public interface RequestContext {
1007
1008     Subject getSecuritySubject();
1009
1010     String getServiceName();

```

```

1011     <CB> ServiceReference<CB> getCallbackReference();
1012     <CB> CB getCallback();
1013     <B> ServiceReference<B> getServiceReference();
1014
1015 }
1016

```

1017 The RequestContext interface has the following methods:

- 1018 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 1019 • **getServiceName()** – Returns the name of the service on the Java implementation the
1020 request came in on
- 1021 • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1022 caller. This method returns null when called for a service request whose interface is not
1023 bidirectional or when called for a callback request.
- 1024 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1025 getCallbackReference() method, this method returns null when called for a service request
1026 whose interface is not bidirectional or when called for a callback request.
- 1027 • **getServiceReference()** – When invoked during the execution of a service operation, this
1028 method MUST return a ServiceReference that represents the service that was invoked.
1029 When invoked during the execution of a callback operation, this method MUST return a
1030 CallableReference that represents the callback that was invoked.

1031 8.3 ServiceReference

1032 ServiceReferences can be injected using the @Reference annotation on a field, a setter method,
1033 or constructor parameter taking the type ServiceReference. The detailed description of the usage
1034 of these methods is described in the section on Asynchronous Programming in this document.

1035 The following Java code defines the **ServiceReference** interface:

```

1036 package org.oasisopen.sca;
1037
1038 public interface ServiceReference<B> extends java.io.Serializable {
1039
1040     B getService();
1041     Class<B> getBusinessInterface();
1042 }
1043

```

1044 The ServiceReference interface has the following methods:

- 1045
- 1046 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
1047 returned is guaranteed to implement the business interface for this reference. The value
1048 returned is a proxy to the target that implements the business interface associated with this
1049 reference.
- 1050 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1051 this reference.

1052 8.4 ServiceRuntimeException

1053 The following snippet shows the **ServiceRuntimeException**.

```

1054
1055 package org.oasisopen.sca;
1056
1057 public class ServiceRuntimeException extends RuntimeException {

```

1058 ...
1059 }
1060
1061

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

1062 **8.5 ServiceUnavailableException**

1063 The following snippet shows the *ServiceUnavailableException*.

```
1064    package org.oasisopen.sca;  
1065  
1066    public class ServiceUnavailableException extends ServiceRuntimeException {  
1067        ...  
1068        }  
1069  
1070
```

1071 This exception signals problems in the interaction with remote services. These are exceptions
1072 that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException
1073 that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since
1074 it most likely requires human intervention

1075 **8.6 InvalidServiceException**

1076 The following snippet shows the *InvalidServiceException*.

```
1077    package org.oasisopen.sca;  
1078  
1079    public class InvalidServiceException extends ServiceRuntimeException {  
1080        ...  
1081        }  
1082  
1083
```

1084 This exception signals that the ServiceReference is no longer valid. This can happen when the
1085 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1086 be resolved by retrying the operation and will most likely require human intervention.

1087 **8.7 Constants**

1088 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java
1089 APIs and Annotations. The following snippet shows the Constants interface:

```
1090    package org.oasisopen.sca;  
1091  
1092    public interface Constants {  
1093        String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";  
1094        String SCA_PREFIX = "{"+SCA_NS+"}";  
1095        }  
1096
```


1097

9 Java Annotations

1098

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

1099

1100

1101

1102

1103

1104

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

1105

1106

1107

SCA annotations are not allowed on static methods and 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.

1108

9.1 @AllowsPassByReference

1109

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

1110

1111

1112

1113

1114

1115

1116

1117

1118

1119

1120

1121

1122

1123

1124

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

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

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

1125

1126

1127

1128

1129

1130

1131

1132

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

1133

`@AllowsPassByReference` has no attributes

1134

1135

1136

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

1137

1138

1139

1140

1141

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

1142 9.2 @Authentication

1143 The following Java code defines the **@Authentication** annotation:

```
1144 package org.oasisopen.sca.annotation;
1145
1146 import static java.lang.annotation.ElementType.FIELD;
1147 import static java.lang.annotation.ElementType.METHOD;
1148 import static java.lang.annotation.ElementType.PARAMETER;
1149 import static java.lang.annotation.ElementType.TYPE;
1150 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1151 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1152
1153 import java.lang.annotation.Inherited;
1154 import java.lang.annotation.Retention;
1155 import java.lang.annotation.Target;
1156
1157 @Inherited
1158 @Target({TYPE, FIELD, METHOD, PARAMETER})
1159 @Retention(RUNTIME)
1160 @Intent(Authentication.AUTHENTICATION)
1161 public @interface Authentication {
1162     String AUTHENTICATION = SCA_PREFIX + "authentication";
1163     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
1164     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";
1165
1166     /**
1167      * List of authentication qualifiers (such as "message"
1168      * or "transport").
1169      *
1170      * @return authentication qualifiers
1171      */
1172     @Qualifier
1173     String[] value() default "";
1174 }
1175
```

1176 The **@Authentication** annotation is used to indicate that the invocation requires authentication.
1177 See the [section on Application of Intent Annotations](#) for samples and details.

1178 9.3 @Callback

1179 The following Java code defines shows the **@Callback** annotation:

```
1180
1181 package org.oasisopen.sca.annotation;
1182
1183 import static java.lang.annotation.ElementType.TYPE;
1184 import static java.lang.annotation.ElementType.METHOD;
1185 import static java.lang.annotation.ElementType.FIELD;
1186 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1187 import java.lang.annotation.Retention;
1188 import java.lang.annotation.Target;
1189
1190 @Target(TYPE, METHOD, FIELD)
1191 @Retention(RUNTIME)
1192 public @interface Callback {
1193
1194     Class<?> value() default Void.class;

```

1195 }

1196

1197

1198 The @Callback annotation is used to annotate a service interface with a callback interface, which
1199 takes the Java Class object of the callback interface as a parameter.

1200 The @Callback annotation has the following attribute:

- 1201 • **value** – the name of a Java class file containing the callback interface

1202

1203 The @Callback annotation can also be used to annotate a method or a field of an SCA
1204 implementation class, in order to have a callback object injected

1205

1206 The following snippet shows a @Callback annotation on an interface:

1207

```
1208 @Remotable  
1209 @Callback(MyServiceCallback.class)  
1210 public interface MyService {  
1211  
1212     void someAsyncMethod(String arg);  
1213 }  
1214
```

1215

1215 An example use of the @Callback annotation to declare a callback interface follows:

1216

```
1217 package somepackage;  
1218 import org.oasisopen.sca.annotation.Callback;  
1219 import org.oasisopen.sca.annotation.Remotable;  
1220 @Remotable  
1221 @Callback(MyServiceCallback.class)  
1222 public interface MyService {  
1223  
1224     void someMethod(String arg);  
1225 }  
1226  
1227 @Remotable  
1228 public interface MyServiceCallback {  
1229  
1230     void receiveResult(String result);  
1231 }  
1232
```

1232

1233 In this example, the implied component type is:

1234

```
1235 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
1236  
1237     <service name="MyService">  
1238         <interface.java interface="somepackage.MyService"  
1239             callbackInterface="somepackage.MyServiceCallback"/>  
1240     </service>  
1241 </componentType>
```

1242 9.4 @ComponentName

1243 The following Java code defines the @ComponentName annotation:

```

1244
1245 package org.oasisopen.sca.annotation;
1246
1247 import static java.lang.annotation.ElementType.METHOD;
1248 import static java.lang.annotation.ElementType.FIELD;
1249 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1250 import java.lang.annotation.Retention;
1251 import java.lang.annotation.Target;
1252
1253 @Target({METHOD, FIELD})
1254 @Retention(RUNTIME)
1255 public @interface ComponentName {
1256
1257 }
1258

```

1259 The @ComponentName annotation is used to denote a Java class field or setter method that is
1260 used to inject the component name.

1261 The following snippet shows a component name field definition sample.

```

1262
1263 @ComponentName
1264 private String componentName;
1265

```

1266 The following snippet shows a component name setter method sample.

```

1267
1268 @ComponentName
1269 public void setComponentName(String name) {
1270     //...
1271 }

```

1272 9.5 @Confidentiality

1273 The following Java code defines the **@Confidentiality** annotation:

```

1274 package org.oasisopen.sca.annotations;
1275
1276 import static java.lang.annotation.ElementType.FIELD;
1277 import static java.lang.annotation.ElementType.METHOD;
1278 import static java.lang.annotation.ElementType.PARAMETER;
1279 import static java.lang.annotation.ElementType.TYPE;
1280 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1281 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1282
1283 import java.lang.annotation.Inherited;
1284 import java.lang.annotation.Retention;
1285 import java.lang.annotation.Target;
1286
1287 @Inherited
1288 @Target({TYPE, FIELD, METHOD, PARAMETER})
1289 @Retention(RUNTIME)
1290 @Intent(Confidentiality.CONFIDENTIALITY)
1291 public @interface Confidentiality {
1292     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1293     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1294     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
1295

```

```

1296
1297     /**
1298      * List of confidentiality qualifiers (such as "message" or
1299      "transport").
1300      *
1301      * @return confidentiality qualifiers
1302      */
1303     @Qualifier
1304     String[] value() default "";
1305 }

```

1306 The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.

1307 See the [section on Application of Intent Annotations](#) for samples and details.

1308 9.6 @Constructor

1309 The following Java code defines the **@Constructor** annotation:

```

1310 package org.oasisopen.sca.annotation;
1311
1312 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1313 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1314 import java.lang.annotation.Retention;
1315 import java.lang.annotation.Target;
1316
1317 @Target(CONSTRUCTOR)
1318 @Retention(RUNTIME)
1319 public @interface Constructor { }
1320
1321

```

1322 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1323 Java component implementation. If this constructor has parameters, each of these parameters
1324 MUST have either a @Property annotation or a @Reference annotation.

1325 The following snippet shows a sample for the @Constructor annotation.

```

1326
1327 public class HelloServiceImpl implements HelloService {
1328
1329     public HelloServiceImpl(){
1330         ...
1331     }
1332
1333     @Constructor
1334     public HelloServiceImpl(@Property(name="someProperty")
1335                             String someProperty ){
1336         ...
1337     }
1338
1339     public String hello(String message) {
1340         ...
1341     }
1342 }

```

1343 9.7 @Context

1344 The following Java code defines the **@Context** annotation:

1345

```

1346 package org.oasisopen.sca.annotation;
1347
1348 import static java.lang.annotation.ElementType.METHOD;
1349 import static java.lang.annotation.ElementType.FIELD;
1350 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1351 import java.lang.annotation.Retention;
1352 import java.lang.annotation.Target;
1353
1354 @Target({METHOD, FIELD})
1355 @Retention(RUNTIME)
1356 public @interface Context {
1357
1358 }
1359

```

1360 The @Context annotation is used to denote a Java class field or a setter method that is used to
1361 inject a composite context for the component. The type of context to be injected is defined by the
1362 type of the Java class field or type of the setter method input argument; the type is either
1363 **ComponentContext** or **RequestContext**.

1364 The @Context annotation has no attributes.

1365

1366 The following snippet shows a ComponentContext field definition sample.

1367

```

1368 @Context
1369 protected ComponentContext context;
1370

```

1371 The following snippet shows a RequestContext field definition sample.

1372

```

1373 @Context
1374 protected RequestContext context;

```

1375 9.8 @Destroy

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

1377

```

1378 package org.oasisopen.sca.annotation;
1379
1380 import static java.lang.annotation.ElementType.METHOD;
1381 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1382 import java.lang.annotation.Retention;
1383 import java.lang.annotation.Target;
1384
1385 @Target(METHOD)
1386 @Retention(RUNTIME)
1387 public @interface Destroy {
1388
1389 }
1390

```

1391 The @Destroy annotation is used to denote a single Java class method that will be called when the
1392 scope defined for the implementation class ends. The method MAY have any access modifier and
1393 MUST have a void return type and no arguments.

1394 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1395 when the scope defined for the implementation class ends. If the implementation class has a

1396 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1397 NOT instantiate the implementation class.

1398 The following snippet shows a sample for a destroy method definition.

1399

```
1400 @Destroy  
1401 public void myDestroyMethod() {  
1402     ...  
1403 }
```

1404 9.9 @EagerInit

1405 The following Java code defines the **@EagerInit** annotation:

1406

```
1407 package org.oasisopen.sca.annotation;  
1408  
1409 import static java.lang.annotation.ElementType.TYPE;  
1410 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1411 import java.lang.annotation.Retention;  
1412 import java.lang.annotation.Target;  
1413  
1414 @Target(TYPE)  
1415 @Retention(RUNTIME)  
1416 public @interface EagerInit {  
1417  
1418 }  
1419
```

1420 The **@EagerInit** annotation is used to annotate the Java class of a COMPOSITE scoped
1421 implementation for eager initialization. When marked for eager initialization, the composite scoped
1422 instance is created when its containing component is started.

1423 9.10 @Init

1424 The following Java code defines the **@Init** annotation:

1425

```
1426 package org.oasisopen.sca.annotation;  
1427  
1428 import static java.lang.annotation.ElementType.METHOD;  
1429 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1430 import java.lang.annotation.Retention;  
1431 import java.lang.annotation.Target;  
1432  
1433 @Target(METHOD)  
1434 @Retention(RUNTIME)  
1435 public @interface Init {  
1436  
1437 }  
1438 }  
1439
```

1440 The @Init annotation is used to denote a single Java class method that is called when the scope
1441 defined for the implementation class starts. The method MAY have any access modifier and MUST
1442 have a void return type and no arguments.

1443 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1444 after all property and reference injection is complete. If the implementation class has a method

1445 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1446 instantiate the implementation class.

1447 The following snippet shows an example of an init method definition.

1448

```
1449 @Init
1450 public void myInitMethod() {
1451     ...
1452 }
```

1453 9.11 @Integrity

1454 The following Java code defines the **@Integrity** annotation:

1455

```
1456 package org.oasisopen.sca.annotation;
```

1457

```
1458 import static java.lang.annotation.ElementType.FIELD;
1459 import static java.lang.annotation.ElementType.METHOD;
1460 import static java.lang.annotation.ElementType.PARAMETER;
1461 import static java.lang.annotation.ElementType.TYPE;
1462 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1463 import static org.oasisopen.Constants.SCA_PREFIX;
```

1464

```
1465 import java.lang.annotation.Inherited;
```

```
1466 import java.lang.annotation.Retention;
```

```
1467 import java.lang.annotation.Target;
```

1468

```
1469 @Inherited
```

```
1470 @Target({TYPE, FIELD, METHOD, PARAMETER})
```

```
1471 @Retention(RUNTIME)
```

```
1472 @Intent(Integrity.INTEGRITY)
```

```
1473 public @interface Integrity {
```

```
1474     String INTEGRITY = SCA_PREFIX + "integrity";
```

```
1475     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
```

```
1476     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
```

1477

```
1478     /**
```

```
1479      * List of integrity qualifiers (such as "message" or "transport").
```

```
1480      *
```

```
1481      * @return integrity qualifiers
```

```
1482      */
```

```
1483     @Qualifier
```

```
1484     String[] value() default "";
```

```
1485 }
```

1486

1487 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no
1488 tampering of the messages between client and service).

1489 See the [section on Application of Intent Annotations](#) for samples and details.

1490 9.12 @Intent

1491 The following Java code defines the **@Intent** annotation:

1492

```
1493 package org.oasisopen.sca.annotation;
```

1494

```
1495 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
```



```

1496     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1497     import java.lang.annotation.Retention;
1498     import java.lang.annotation.Target;
1499
1500     @Target({ANNOTATION_TYPE})
1501     @Retention(RUNTIME)
1502     public @interface Intent {
1503         /**
1504          * The qualified name of the intent, in the form defined by
1505          * {@link javax.xml.namespace.QName#toString}.
1506          * @return the qualified name of the intent
1507          */
1508         String value() default "";
1509
1510         /**
1511          * The XML namespace for the intent.
1512          * @return the XML namespace for the intent
1513          */
1514         String targetNamespace() default "";
1515
1516         /**
1517          * The name of the intent within its namespace.
1518          * @return name of the intent within its namespace
1519          */
1520         String localPart() default "";
1521     }
1522

```

1523 The @Intent annotation is used for the creation of new annotations for specific intents. It is not
1524 expected that the @Intent annotation will be used in application code.

1525 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
1526 define new intent annotations.

1527 9.13 @OneWay

1528 The following Java code defines the **@OneWay** annotation:

```

1529
1530     package org.oasisopen.sca.annotation;
1531
1532     import static java.lang.annotation.ElementType.METHOD;
1533     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1534     import java.lang.annotation.Retention;
1535     import java.lang.annotation.Target;
1536
1537     @Target(METHOD)
1538     @Retention(RUNTIME)
1539     public @interface OneWay {
1540
1541     }
1542
1543

```

1544 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1545 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1546 Programming.

1547 The @OneWay annotation has no attributes.

1548 The following snippet shows the use of the @OneWay annotation on an interface.

```

1549 package services.hello;
1550
1551 import org.oasisopen.sca.annotation.OneWay;
1552
1553 public interface HelloService {
1554     @OneWay
1555     void hello(String name);
1556 }

```

9.14 @PolicySets

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

```

1559 package org.oasisopen.sca.annotation;
1560
1561 import static java.lang.annotation.ElementType.FIELD;
1562 import static java.lang.annotation.ElementType.METHOD;
1563 import static java.lang.annotation.ElementType.PARAMETER;
1564 import static java.lang.annotation.ElementType.TYPE;
1565 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1566
1567 import java.lang.annotation.Retention;
1568 import java.lang.annotation.Target;
1569
1570 @Target({TYPE, FIELD, METHOD, PARAMETER})
1571 @Retention(RUNTIME)
1572 public @interface PolicySets {
1573     /**
1574      * Returns the policy sets to be applied.
1575      *
1576      * @return the policy sets to be applied
1577      */
1578     String[] value() default "";
1579 }
1580
1581

```

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

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

9.15 @Property

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

```

1587 package org.oasisopen.sca.annotation;
1588
1589 import static java.lang.annotation.ElementType.METHOD;
1590 import static java.lang.annotation.ElementType.FIELD;
1591 import static java.lang.annotation.ElementType.PARAMETER;
1592 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1593 import java.lang.annotation.Retention;
1594 import java.lang.annotation.Target;
1595
1596 @Target({METHOD, FIELD, PARAMETER})
1597 @Retention(RUNTIME)
1598 public @interface Property {
1599     String name() default "";
1600 }

```

```
1601     boolean required() default true;
1602 }
1603
```

1604 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1605 parameter that is used to inject an SCA property value. The type of the property injected, which
1606 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1607 the type of the input parameter of the setter method or constructor.

1608 The @Property annotation can be used on fields, on setter methods or on a constructor method
1609 parameter. However, the @Property annotation MUST NOT be used on a class field that is declared
1610 as final.

1611 Properties can also be injected via setter methods even when the @Property annotation is not
1612 present. However, the @Property annotation must be used in order to inject a property onto a
1613 non-public field. In the case where there is no @Property annotation, the name of the property is
1614 the same as the name of the field or setter.

1615 Where there is both a setter method and a field for a property, the setter method is used.

1616 The @Property annotation has the following attributes:

- 1617 • **name (optional)** – the name of the property. For a field annotation, the default is the
1618 name of the field of the Java class. For a setter method annotation, the default is the
1619 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1620 constructor parameter annotation, there is no default and the name attribute MUST be
1621 present.
- 1622 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1623 constructor parameter annotation, this attribute MUST have the value true.

1624

1625 The following snippet shows a property field definition sample.

1626

```
1627 @Property(name="currency", required=true)
1628 protected String currency;
```

1629

1630 The following snippet shows a property setter sample

1631

```
1632 @Property(name="currency", required=true)
1633 public void setCurrency( String theCurrency ) {
1634     ....
1635 }
```

1636

1637 If the property is defined as an array or as any type that extends or implements
1638 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to
1639 true.

1640 The following snippet shows the definition of a configuration property using the @Property
1641 annotation for a collection.

1642

```
1643 ...
1644 private List<String> helloConfigurationProperty;
1645 @Property(required=true)
1646 public void setHelloConfigurationProperty(List<String> property) {
1647     helloConfigurationProperty = property;
1648 }
```

1649 ...

1650 9.16 @Qualifier

1651 The following Java code defines the **@Qualifier** annotation:

```
1652 package org.oasisopen.sca.annotation;
1653
1654 import static java.lang.annotation.ElementType.METHOD;
1655 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1656
1657 import java.lang.annotation.Retention;
1658 import java.lang.annotation.Target;
1659
1660 @Target(METHOD)
1661 @Retention(RUNTIME)
1662 public @interface Qualifier {
1663 }
1664
1665
```

1666 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition, defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.

1670 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to define new intent annotations.

1672 9.17 @Reference

1673 The following Java code defines the **@Reference** annotation:

```
1674
1675 package org.oasisopen.sca.annotation;
1676
1677 import static java.lang.annotation.ElementType.METHOD;
1678 import static java.lang.annotation.ElementType.FIELD;
1679 import static java.lang.annotation.ElementType.PARAMETER;
1680 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1681 import java.lang.annotation.Retention;
1682 import java.lang.annotation.Target;
1683 @Target({METHOD, FIELD, PARAMETER})
1684 @Retention(RUNTIME)
1685 public @interface Reference {
1686
1687     String name() default "";
1688     boolean required() default true;
1689 }
1690
```

1691 The @Reference annotation type is used to annotate a Java class field, a setter method, or a constructor parameter that is used to inject a service that resolves the reference. The interface of the service injected is defined by the type of the Java class field or the type of the input parameter of the setter method or constructor.

1695 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1696 References can also be injected via setter methods even when the @Reference annotation is not present. However, the @Reference annotation must be used in order to inject a reference onto a non-public field. In the case where there is no @Reference annotation, the name of the reference is the same as the name of the field or setter.

1700 Where there is both a setter method and a field for a reference, the setter method is used.

1701 The @Reference annotation has the following attributes:

- 1702 • **name (optional)** – the name of the reference. For a field annotation, the default is the
1703 name of the field of the Java class. For a setter method annotation, the default is the
1704 JavaBeans property name corresponding to the setter method name. For a constructor
1705 parameter annotation, there is no default and the name attribute MUST be present.
- 1706 • **required (optional)** – whether injection of service or services is required. Defaults to true.
1707 For a constructor parameter annotation, this attribute MUST have the value true.

1708

1709 The following snippet shows a reference field definition sample.

1710

```
1711 @Reference(name="stockQuote", required=true)  
1712 protected StockQuoteService stockQuote;
```

1713

1714 The following snippet shows a reference setter sample

1715

```
1716 @Reference(name="stockQuote", required=true)  
1717 public void setStockQuote( StockQuoteService theSQService ) {  
1718     ...  
1719 }
```

1720

1721 The following fragment from a component implementation shows a sample of a service reference
1722 using the @Reference annotation. The name of the reference is "helloService" and its type is
1723 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1724 helloService reference.

1725

```
1726 package services.hello;  
1727  
1728 private HelloService helloService;  
1729  
1730 @Reference(name="helloService", required=true)  
1731 public setHelloService(HelloService service) {  
1732     helloService = service;  
1733 }  
1734  
1735 public void clientMethod() {  
1736     String result = helloService.hello("Hello World!");  
1737     ...  
1738 }  
1739
```

1740 The presence of a @Reference annotation is reflected in the componentType information that the
1741 runtime generates through reflection on the implementation class. The following snippet shows
1742 the component type for the above component implementation fragment.

1743

```
1744 <?xml version="1.0" encoding="ASCII"?>  
1745 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1746  
1747     <!-- Any services offered by the component would be listed here -->
```

```

1748     <reference name="helloService" multiplicity="1..1">
1749         <interface.java interface="services.hello.HelloService"/>
1750     </reference>
1751
1752 </componentType>
1753

```

1754 If the reference is not an array or collection, then the implied component type has a reference
1755 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1756 attribute – 1..1 applies if required=true.

1757
1758 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,
1759 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending
1760 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1761 required=true.

1762
1763 The following fragment from a component implementation shows a sample of a service reference
1764 definition using the @Reference annotation on a java.util.List. The name of the reference is
1765 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1766 services referenced by the helloServices reference. In this case, at least one HelloService should
1767 be present, so **required** is true.

```

1768     @Reference(name="helloServices", required=true)
1769     protected List<HelloService> helloServices;
1770
1771     public void clientMethod() {
1772
1773         ...
1774         for (int index = 0; index < helloServices.size(); index++) {
1775             HelloService helloService =
1776                 (HelloService)helloServices.get(index);
1777             String result = helloService.hello("Hello World!");
1778         }
1779         ...
1780     }
1781
1782

```

1783 The following snippet shows the XML representation of the component type reflected from for the
1784 former component implementation fragment. There is no need to author this component type in
1785 this case since it can be reflected from the Java class.

```

1786
1787 <?xml version="1.0" encoding="ASCII"?>
1788 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1789
1790     <!-- Any services offered by the component would be listed here -->
1791     <reference name="helloServices" multiplicity="1..n">
1792         <interface.java interface="services.hello.HelloService"/>
1793     </reference>
1794
1795 </componentType>
1796

```

1797 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1798 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1799 of 0..N must be an empty array or collection.

1800 **9.17.1 Reinjection**

1801 References MAY be reinjected after the initial creation of a component if the reference target
 1802 changes due to a change in wiring that has occurred since the component was initialized. In order
 1803 for reinjection to occur, the following MUST be true:

- 1804 1. The component MUST NOT be STATELESS scoped.
- 1805 2. The reference MUST use either field-based injection or setter injection. References that are
 1806 injected through constructor injection MUST NOT be changed. Setter injection allows for
 1807 code in the setter method to perform processing in reaction to a change.

1808 If a reference target changes and the reference is not reinjected, the reference MUST continue to
 1809 work as if the reference target was not changed.

1810 If an operation is called on a reference where the target of that reference has been undeployed,
 1811 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
 1812 where the target of the reference has become unavailable for some reason, the SCA runtime
 1813 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
 1814 reference MAY continue to work, depending on the runtime and the type of change that was made.
 1815 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1816 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
 1817 corresponds to the reference that is passed as a parameter to cast(). If the reference is
 1818 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
 1819 to work as if the reference target was not changed. If the target of a ServiceReference has been
 1820 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
 1821 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
 1822 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
 1823 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
 1824 work, depending on the runtime and the type of change that was made. If it doesn't work, the
 1825 exception thrown will depend on the runtime and the cause of the failure.

1826 A reference or ServiceReference accessed through the component context by calling getService()
 1827 or getServiceReference() MUST correspond to the current configuration of the domain. This
 1828 applies whether or not reinjection has taken place. If the target has been undeployed or has
 1829 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
 1830 and attempts to call business methods SHOULD throw an exception as described above. If the
 1831 target has changed, the result SHOULD be a reference to the changed service.

1832 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
 1833 means that in the cases listed above where reference reinjection is not allowed, the array or
 1834 Collection for the reference MUST NOT change its contents. In cases where the contents of a
 1835 reference collection MAY change, then for references that use setter injection, the setter method
 1836 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
 1837 the same array or Collection object previously injected to the component.

1838

| | Effect on | | |
|---------------------------------------|--|---|--|
| Change event | Reference | Existing ServiceReference Object | Subsequent invocations of ComponentContext.getServiceReference() or getService() |
| Change to the target of the reference | MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed. | MUST continue to work as if the reference target was not changed. | Result corresponds to the current configuration of the domain. |
| Target service | Business methods SHOULD throw | Business methods SHOULD throw | Result SHOULD be a reference to the undeployed |

| | | | |
|---|--|--|--|
| undeployed | InvalidServiceException. | InvalidServiceException. | or unavailable service. Business methods SHOULD throw InvalidServiceException. |
| Target service changed | MAY 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. | MAY 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 SHOULD be a reference to the changed service. |
| <p>* Other conditions:</p> <ol style="list-style-type: none"> 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. <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p> | | | |

1839

1840

9.18 @Remotable

1841

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

1842

1843

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

1844

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

1846

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

1847

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

1848

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

1849

1850

```
@Target (TYPE)
```

1852

```
@Retention(RUNTIME)
```

1853

```
public @interface Remotable {
```

1854

```
}
```

1855

1856

1857

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.

1858

1859

The @Remotable annotation has no attributes.

1860

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

1861

1862

```
package services.hello;
```

1863

```
import org.oasisopen.sca.annotation.*;
```

1865

```
@Remotable
```

1866

```
public interface HelloService {
```

1867

```
String hello(String message);
```

1868

1869

1870 }
1871

1872 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
1873 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1874 Complex data types exchanged via remotable service interfaces MUST be compatible with the
1875 marshalling technology used by the service binding. For example, if the service is going to be
1876 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types
1877 or Service Data Objects (SDOs) [SDO].

1878 Independent of whether the remotable service is called from outside of the composite that
1879 contains it or from another component in the same composite, the data exchange semantics are
1880 **by-value**.

1881 Implementations of remotable services can modify input data during or after an invocation and
1882 can modify return data after the invocation. If a remotable service is called locally or remotely, the
1883 SCA container is responsible for making sure that no modification of input data or post-invocation
1884 modifications to return data are seen by the caller.

1885 The following snippet shows a remotable Java service interface.

```
1886  
1887 package services.hello;  
1888  
1889 import org.oasisopen.sca.annotation.*;  
1890  
1891 @Remotable  
1892 public interface HelloService {  
1893     String hello(String message);  
1894 }  
1895  
1896 package services.hello;  
1897  
1898 import org.oasisopen.sca.annotation.*;  
1899  
1900 @Service(HelloService.class)  
1901 public class HelloServiceImpl implements HelloService {  
1902     public String hello(String message) {  
1903         ...  
1904     }  
1905 }  
1906 }  
1907
```

1908 9.19 @Requires

1909 The following Java code defines the **@Requires** annotation:

```
1910 package org.oasisopen.sca.annotation;  
1911  
1912 import static java.lang.annotation.ElementType.FIELD;  
1913 import static java.lang.annotation.ElementType.METHOD;  
1914 import static java.lang.annotation.ElementType.PARAMETER;  
1915 import static java.lang.annotation.ElementType.TYPE;  
1916 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1917  
1918 import java.lang.annotation.Inherited;  
1919 import java.lang.annotation.Retention;  
1920 import java.lang.annotation.Target;
```

```

1922
1923 @Inherited
1924 @Retention(RUNTIME)
1925 @Target({TYPE, METHOD, FIELD, PARAMETER})
1926 public @interface Requires {
1927     /**
1928      * Returns the attached intents.
1929      *
1930      * @return the attached intents
1931      */
1932     String[] value() default "";
1933 }
1934

```

1935 The **@Requires** annotation supports general purpose intents specified as strings. Users can also
1936 define specific intent annotations using the @Intent annotation.

1937 See the [section "General Intent Annotations"](#) for details and samples.

1938 9.20 @Scope

1939 The following Java code defines the **@Scope** annotation:

```

1940 package org.oasisopen.sca.annotation;
1941
1942 import static java.lang.annotation.ElementType.TYPE;
1943 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1944 import java.lang.annotation.Retention;
1945 import java.lang.annotation.Target;
1946
1947 @Target(TYPE)
1948 @Retention(RUNTIME)
1949 public @interface Scope {
1950
1951     String value() default "STATELESS";
1952 }

```

1953 The @Scope annotation MUST only be used on a service's implementation class. It is an error to
1954 use this annotation on an interface.

1955 The @Scope annotation has the following attribute:

- 1956 • **value** – the name of the scope.
- 1957 For 'STATELESS' implementations, a different implementation instance can be used to
- 1958 service each request. Implementation instances can be newly created or be drawn from a
- 1959 pool of instances.
- 1960 SCA defines the following scope names, but others can be defined by particular Java-
- 1961 based implementation types:
- 1962 STATELESS
- 1963 COMPOSITE

1964 The default value is STATELESS.

1965 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```

1966 package services.hello;
1967
1968 import org.oasisopen.sca.annotation.*;
1969
1970 @Service(HelloService.class)
1971 @Scope("COMPOSITE")
1972 public class HelloServiceImpl implements HelloService {
1973
1974     public String hello(String message) {

```

```
1975     ...
1976     }
1977 }
1978
```

1979 9.21 @Service

1980 The following Java code defines the **@Service** annotation:

```
1981 package org.oasisopen.sca.annotation;
1982
1983 import static java.lang.annotation.ElementType.TYPE;
1984 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1985 import java.lang.annotation.Retention;
1986 import java.lang.annotation.Target;
1987
1988 @Target(TYPE)
1989 @Retention(RUNTIME)
1990 public @interface Service {
1991     Class<?>[] interfaces() default {};
1992     Class<?> value() default Void.class;
1993 }
1994
1995
```

1996 The @Service annotation is used on a component implementation class to specify the SCA services
1997 offered by the implementation. The class need not be declared as implementing all of the
1998 interfaces implied by the services, but all methods of the service interfaces must be present. A
1999 class used as the implementation of a service is not required to have a @Service annotation. If a
2000 class has no @Service annotation, then the rules determining which services are offered and what
2001 interfaces those services have are determined by the specific implementation type.

2002 The @Service annotation has the following attributes:

- 2003 • **interfaces** – The value is an array of interface or class objects that should be exposed as
2004 services by this component.
- 2005 • **value** – A shortcut for the case when the class provides only a single service interface.

2006 Only one of these attributes should be specified.

2007

2008 A @Service annotation with no attributes is meaningless, it is the same as not having the
2009 annotation there at all.

2010 The **service names** of the defined services default to the names of the interfaces or class, without
2011 the package name.

2012 A component MUST NOT have two services with the same Java simple name. If a Java
2013 implementation needs to realize two services with the same Java simple name then this can be
2014 achieved through subclassing of the interface.

2015 The following snippet shows an implementation of the HelloService marked with the @Service
2016 annotation.

```
2017 package services.hello;
2018
2019 import org.oasisopen.sca.annotation.Service;
2020
2021 @Service(HelloService.class)
2022 public class HelloServiceImpl implements HelloService {
2023
2024     public void hello(String name) {
```

```
2025         System.out.println("Hello " + name);
2026     }
2027 }
2028
```

2029 10 WSDL to Java and Java to WSDL

2030 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
2031 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java
2032 interfaces from WSDL portTypes and vice versa.

2033 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a
2034 @WebService annotation on the class, even if it doesn't, and the
2035 @org.oasisopen.sca.annotation.OneWay annotation should be treated as a synonym for the
2036 @javax.jws.OneWay annotation. For the WSDL-to-Java mapping, the generated @WebService
2037 annotation implies that the interface is @Remotable.

2038 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
2039 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
2040 and MAY support the SDO 2.1 mapping. Having a choice of binding technologies is allowed, as
2041 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is
2042 referenced by the JAX-WS specification.

2043 The JAX-WS mappings are applied with the following restrictions:

- 2044 • No support for holders

2045
2046 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
2047 model is used.

2048 10.1 JAX-WS Client Asynchronous API for a Synchronous Service

2049 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
2050 application with a means of invoking that service asynchronously, so that the client can invoke a service
2051 operation and proceed to do other work without waiting for the service operation to complete its
2052 processing. The client application can retrieve the results of the service either through a polling
2053 mechanism or via a callback method which is invoked when the operation completes.

2054 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional
2055 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces
2056 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are
2057 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the
2058 Assembly specification. These methods are recognized as follows.

2059 For each method M in the interface, if another method P in the interface has

- 2060 a. a method name that is M's method name with the characters "Async" appended, and
- 2061 b. the same parameter signature as M, and
- 2062 c. a return type of Response<R> where R is the return type of M

2063 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

2064 For each method M in the interface, if another method C in the interface has

- 2065 a. a method name that is M's method name with the characters "Async" appended, and
- 2066 b. a parameter signature that is M's parameter signature with an additional final parameter of type
2067 AsyncHandler<R> where R is the return type of M, and
- 2068 c. a return type of Future<?>

2069 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

2070 As an example, an interface can be defined in WSDL as follows:

```
2071 <!-- WSDL extract -->  
2072 <message name="getPrice">
```

```
2073 <part name="ticker" type="xsd:string"/>
2074 </message>
2075
2076 <message name="getPriceResponse">
2077 <part name="price" type="xsd:float"/>
2078 </message>
2079
2080 <portType name="StockQuote">
2081 <operation name="getPrice">
2082 <input message="tns:getPrice"/>
2083 <output message="tns:getPriceResponse"/>
2084 </operation>
2085 </portType>
```

2086

2087 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2088 // asynchronous mapping
2089 @WebService
2090 public interface StockQuote {
2091     float getPrice(String ticker);
2092     Response<Float> getPriceAsync(String ticker);
2093     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2094 }
```

2095

2096 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2097 // synchronous mapping
2098 @WebService
2099 public interface StockQuote {
2100     float getPrice(String ticker);
2101 }
```

2102

2103 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2104 example, if the client implementation uses the asynchronous form of the interface, the two
2105 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2106 WS specification.

2107

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

```
2108 <?xml version="1.0" encoding="UTF-8"?>
2109 <!-- (c) Copyright SCA Collaboration 2006 -->
2110 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2111         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2112         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2113         elementFormDefault="qualified">
2114
2115     <include schemaLocation="sca-core.xsd"/>
2116
2117     <element name="interface.java" type="sca:JavaInterface"
2118             substitutionGroup="sca:interface"/>
2119     <complexType name="JavaInterface">
2120         <complexContent>
2121             <extension base="sca:Interface">
2122                 <sequence>
2123                     <any namespace="##other" processContents="lax"
2124                         minOccurs="0" maxOccurs="unbounded"/>
2125                 </sequence>
2126                 <attribute name="interface" type="NCName" use="required"/>
2127                 <attribute name="callbackInterface" type="NCName"
2128                     use="optional"/>
2129                 <anyAttribute namespace="##any" processContents="lax"/>
2130             </extension>
2131         </complexContent>
2132     </complexType>
2133 </schema>
```

2134

B. Conformance Items

2135 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2136 specification.

2137

| Conformance ID | Description |
|----------------|--|
| [JCA30001] | @interface MUST be the fully qualified name of the Java interface class |
| [JCA30002] | @callbackInterface MUST be the fully qualified name of a Java interface used for callbacks |
| [JCA30003] | However, if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class. |

2138

2139 **C. Acknowledgements**

2140 The following individuals have participated in the creation of this specification and are gratefully
2141 acknowledged:

2142 **Participants:**

2143 [Participant Name, Affiliation | Individual Member]

2144 [Participant Name, Affiliation | Individual Member]

2145

D. Non-Normative Text

2147

E. Revision History

2148 [optional; should not be included in OASIS Standards]

2149

| 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. |

2150