OASIS

Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 01, Revision 4a3 + Issue 27

12086 December January 2008

Specification URIs:
This Version:

http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev42.html
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev24.doc
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev42.pdf

	Field Code Changed
	Field Code Changed
	Field Code Changed

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 may 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 and conversational 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 may 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 may 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

Table of Contents

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 may 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 and conversational 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 may 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^{TM} 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, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.
[ASSEMBLY]	SCA Assembly Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf
[SDO]	SDO 2.1 Specification, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf
[JAX-B]	JAXB 2.1 Specification, http://www.jcp.org/en/jsr/detail?id=222
[WSDL]	WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsdl, WSDL 2.0: http://www.w3.org/TR/wsdl20/

43	**[POLICY]**	SCA Policy Framework,
44		http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf
45	**[JSR-250]**	Common Annotation for Java Platform specification (JSR-250),
46		http://www.jcp.org/en/jsr/detail?id=250
47	**[JAX-WS]**	JAX-WS 2.1 Specification (JSR-224),
48		http://www.jcp.org/en/jsr/detail?id=224
49	**[JAVABEANS]**	JavaBeans 1.01 Specification,
50		http://java.sun.com/javase/technologies/desktop/javabeans/api/
51		

1.3 Non-Normative References

| 53 | **TBD** | TBD |

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.

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

96 2.1.4 @Reference

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

100 2.1.5 @Property

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

104 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

111   This document defines three scopes:

112   •   STATELESS

113   •   CONVERSATION

114   •   COMPOSITE

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

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

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

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

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

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

127

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

```
136            }
137
```

138   The following sections specify four standard scopes, which a Java-based implementation type may
139   support.

2.2.1 Stateless scope

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

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

2.2.2 Composite scope

150   All service requests are dispatched to the same implementation instance for the lifetime of the
151   containing composite. The lifetime of the containing composite is defined as the time it becomes
152   active in the runtime to the time it is deactivated, either normally or abnormally.

153   A composite scoped implementation may also specify eager initialization using the **@EagerInit**
154   annotation. When marked for eager initialization, the composite scoped instance is created when
155   its containing component is started. If a method is marked with the @Init annotation, it is called
156   when the instance is created.

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

2.2.3 Conversation scope

161   A **conversation** is defined as a series of correlated interactions between a client and a target
162   service. A conversational scope starts when the first service request is dispatched to an
163   implementation instance offering a conversational service. A conversational scope completes after
164   an end operation defined by the service contract is called and completes processing or the
165   conversation expires. A conversation may be long-running (for example, hours, days or weeks)
166   and the SCA runtime may choose to passivate implementation instances. If this occurs, the
167   runtime must guarantee that implementation instance state is preserved.

168   Note that in the case where a conversational service is implemented by a Java class marked as
169   conversation scoped, the SCA runtime will transparently handle implementation state. It is also
170   possible for an implementation to manage its own state. For example, a Java class having a
171   stateless (or other) scope could implement a conversational service.

172   A conversational scoped class MUST NOT expose a service using a non-conversational interface.
173   When a service has a conversational interface it MUST be implemented by a conversation-scoped
174   component. If no scope is specified on the implementation, then conversation scope is implied.

175   The concurrency model for the conversation scope is multi-threaded. This means that the SCA
176   runtime MAY run multiple threads in a single conversational scoped implementation instance
177   object and it MUST NOT perform any synchronization.

3 Interface

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

3.1 Java interface element – ("<interface.java>")

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

The following is the pseudo-schema for the interface.java element~~The following snippet shows the schema for the Java interface element.~~

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

The interface.java element has the following attributes:

* *interface (1..1)* – the Java interface class to use for the service interface. @interface MUST be the fully qualified name of the Java interface class~~MUST be the fully qualified name of the Java interface class~~ [JCA30001]

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

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

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

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

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

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

Formatted: French (France)

Comment [ME1]: Issue 101 (and many other changes in this section)

Formatted: Font color: Red

Formatted: Bullets and Numbering

Formatted: Font color: Red

Formatted: Font color: Red

Comment [ME2]: Issue 79

223 ~~For the Java interface type system, *parameters and return types* of the service methods are~~
224 ~~described using Java classes or simple Java types. Service Data Objects [SDO] are the preferred~~
225 ~~form of Java class because of their integration with XML technologies.~~

226

3.2 @Remotable

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

3.3 @Conversational

233 Java service interfaces may be annotated to specify whether their contract is conversational as
234 described in the Assembly Specification [ASSEMBLY] by using the *@Conversational* annotation. A
235 conversational service indicates that requests to the service are correlated in some way.

236 When @Conversational is not specified on a service interface, the service contract is *stateless*.

3.4 @Callback

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

4 Client API

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

4.1 Accessing Services from an SCA Component

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

4.1.1 Using the Component Context API

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

4.2 Accessing Services from non-SCA component implementations

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

4.2.1 ComponentContext

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

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

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

5 Error Handling

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

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

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

6 Asynchronous and Conversational Programming

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

- support for non-blocking method calls

- conversational services

- callbacks

Each of these topics is discussed in the following sections.

Conversational services are services where there is an ongoing sequence of interactions between the client and the service provider, which involve some set of state data – in contrast to the simple case of stateless interactions between a client and a provider. Asynchronous services may often involve the use of a conversation, although this is not mandatory.

6.1 @OneWay

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

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

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

6.2 Conversational Services

A service may be declared as conversational by marking its Java interface with a *@Conversational* annotation. If a service interface is not marked with a @Conversational, it is stateless.

6.2.1 ConversationAttributes

A Java-based implementation class may be marked with a *@ConversationAttributes* annotation, which is used to specify the expiration rules for conversational implementation instances.

An example of the @*ConversationAttributes* is shown below:

```
package com.bigbank;
import org.osoaoasisopen.sca.annotations.ConversationAttributes;

@ConversationAttributes(maxAge="30 days");
public class LoanServiceImpl implements LoanService {

}
```

6.2.2 @EndsConversation

316

317 A method of a conversational interface may be marked with an @EndsConversation annotation.
318 Once a method marked with @EndsConversation has been called, the conversation between client
319 and service provider is at an end, which implies no further methods may be called on that service
320 within the same conversation. This enables both the client and the service provider to free up
321 resources that were associated with the conversation.

322 It is also possible to mark a method on a callback interface (described later) with
323 @EndsConversation, in order for the service provider to be the party that chooses to end the
324 conversation.

325 If a conversation is ended with an explicit outbound call to an @EndsConversation method or
326 through a call to the ServiceReference.endConversation() method, then any subsequent call to an
327 operation on the service reference will start a new conversation. If the conversation ends for any
328 other reason (e.g. a timeout occurred), then until ServiceReference.getConversation().end() is
329 called, the ConversationEndedException is thrown by any conversational operation.

6.3 Passing Conversational Services as Parameters

330

331 The service reference which represents a single conversation can be passed as a parameter to
332 another service, even if that other service is remote. This may be used to allow one component to
333 continue a conversation that had been started by another.

334 A service provider may also create a service reference for itself that it can pass to other services.
335 A service implementation does this with a call to the createSelfReference(…) method:

336
```
        interface ComponentContext{
```
337
```
            …
```
338
```
        <B> ServiceReference<B> createSelfReference(Class
```
339
```
                    businessInterface);
```
340
```
        <B> ServiceReference<B> createSelfReference(Class
```
341
```
                    businessInterface, String serviceName);
```
342
```
        }
```
343

344 The second variant, which takes an additional **serviceName** parameter, must be used if the
345 component implements multiple services.

346 This capability may be used to support complex callback patterns, such as when a callback is
347 applicable only to a subset of a larger conversation. Simple callback patterns are handled by the
348 built-in callback support described later.

6.4 Conversational Client

349

350 The client of a conversational service does not need to be coded in a special way. The client can
351 take advantage of the conversational nature of the interface through the relationship of the
352 different methods in the interface and any data they may share in common. If the service is
353 asynchronous, the client may like to use a feature such as the conversationID to keep track of any
354 state data relating to the conversation.

355 The developer of the client knows that the service is conversational by introspecting the service
356 contract. The following shows how a client accesses the conversational service described above:

357

358
```
    @Reference
```
359
```
    LoanService loanService;
```
360
```
    // Known to be conversational because the interface is marked as
```
361
```
    // conversational
```

```
362     public void applyForMortgage(Customer customer, HouseInfo houseInfo,
363                                  int term)
364     {
365         LoanApplication loanApp;
366         loanApp = createApplication(customer, houseInfo);
367         loanService.apply(loanApp);
368         loanService.lockCurrentRate(term);
369     }
370
371     public boolean isApproved() {
372         return loanService.getLoanStatus().equals("approved");
373     }
374     public LoanApplication createApplication(Customer customer,
375                                              HouseInfo houseInfo) {
376         return …;
377     }
```

6.5 Conversation Lifetime Summary

Starting conversations

Conversations start on the client side when one of the following occur:

- A @Reference to a conversational service is injected
- A call is made to CompositeContext.getServiceReference and then a method of the service is called.

Continuing conversations

The client can continue an existing conversation, by:

- Holding the service reference that was created when the conversation started
- Getting the service reference object passed as a parameter from another service, even remotely
- Loading a service reference that had been written to some form of persistent storage

Ending conversations

A conversation ends, and any state associated with the conversation is freed up, when:

- A service operation that has been annotated @EndsConveration has been called
- The server calls an @EndsConversation method on the @Callback reference
- The server's conversation lifetime timeout occurs
- The client calls Conversation.end()
- Any non-business exception is thrown by a conversational operation

If a method is invoked on a service reference after an @EndsConversation method has been called then a new conversation will automatically be started. If ServiceReference.getConversationID() is called after the @EndsConversation method is called, but before the next conversation has been started, it returns null.

404 If a service reference is used after the service provider's conversation timeout has caused the
405 conversation to be ended, then ConversationEndedException is thrown. In order to use that
406 service reference for a new conversation, its endConversation () method must be called.
407

6.6 Conversation ID

409 Every conversation has a **conversation ID**. The conversation ID can be generated by the system,
410 or it can be supplied by the client component.

411 If a field or setter method is annotated with **@ConversationID**, then the conversation ID for the
412 conversation is injected. The type of the field is not necessarily String. System generated
413 conversation IDs are always strings, but application generated conversation IDs may be other
414 complex types.

6.6.1 Application Specified Conversation IDs

416 It is possible to take advantage of the state management aspects of conversational services while
417 using a client-provided conversation ID. To do this, the client does not use reference injection,
418 but uses the **ServiceReference.setConversationID()** API.

419 The conversation ID that is passed into this method should be an instance of either a String or of
420 an object that is serializable into XML. The ID must be unique to the client component over all
421 time. If the client is not an SCA component, then the ID must be globally unique.

422 Not all conversational service bindings support application-specified conversation IDs or may only
423 support application-specified conversation IDs that are Strings.

6.6.2 Accessing Conversation IDs from Clients

425 Whether the conversation ID is chosen by the client or is generated by the system, the client may
426 access the conversation ID by calling getConversationID() on the current conversation
427 object.

428 If the conversation ID is not application specified, then the
429 ServiceReference.getConversationID() method is only guaranteed to return a valid value
430 after the first operation has been invoked, otherwise it returns null.

6.7 Callbacks

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

436 • an interface for the provided service

437 • a callback interface that must be provided by the client

438 Callbacks may be used for both remotable and local services. Either both interfaces of a
439 bidirectional service must be remotable, or both must be local. It is illegal to mix the two. There
440 are two basic forms of callbacks: stateless callbacks and stateful callbacks.

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

6.7.1 Stateful Callbacks

446 A **stateful** callback represents a specific implementation instance of the component that is the
447 client of the service. The interface of a stateful callback should be marked as **conversational**.

448   The following example interfaces show an interaction over a stateful callback.

```
449    package somepackage;
450    import org.osoaoasisopen.sca.annotations.Callback;
451    import org.osoaoasisopen.sca.annotations.Conversational;
452    import org.osoaoasisopen.sca.annotations.Remotable;
453    @Remotable
454    @Conversational
455    @Callback(MyServiceCallback.class)
456    public interface MyService {
457
458        void someMethod(String arg);
459    }
460
461    @Remotable
462    @Conversational
463    public interface MyServiceCallback {
464
465        void receiveResult(String result);
466    }
467
```

468   An implementation of the service in this example could use the @Callback annotation to request
469   that a stateful callback be injected. The following is a fragment of an implementation of the
470   example service. In this example, the request is passed on to some other component, so that the
471   example service acts essentially as an intermediary. If the example service is conversation
472   scoped, the callback will still be available when the backend service sends back its asynchronous
473   response.

474   When an interface and its callback interface are both marked as conversational, then there is only
475   one conversation that applies in both directions and it has the same lifetime. In this case, if both
476   interfaces declare a @ConversationAttributes annotation, then only the annotation on the main
477   interface applies.

```
478
479    @Callback
480    protected MyServiceCallback callback;
481
482    @Reference
483    protected MyService backendService;
484
485    public void someMethod(String arg) {
486            backendService.someMethod(arg);
487    }
488
489    public void receiveResult(String result) {
490            callback.receiveResult(result);
491    }
492
```

493   This fragment must come from an implementation that offers two services, one that it offers to its
494   clients (MyService) and one that is used for receiving callbacks from the back end
495   (MyServiceCallback). The code snippet below is taken from the client of this service, which also
496   implements the methods defined in MyServiceCallback.
497

```
498
499         private MyService myService;
500
501         @Reference
502         public void setMyService(MyService service) {
503                 myService = service;
504         }
505
506         public void aClientMethod() {
507                 ...
508             myService.someMethod(arg);
509         }
510
511         public void receiveResult(String result) {
512                 // code to process the result
513         }
514
```

Stateful callbacks support some of the same use cases as are supported by the ability to pass service references as parameters. The primary difference is that stateful callbacks do not require any additional parameters be passed with service operations. This can be a great convenience. If the service has many operations and any of those operations could be the first operation of the conversation, it would be unwieldy to have to take a callback parameter as part of every operation, just in case it is the first operation of the conversation. It is also more natural than requiring application developers to invoke an explicit operation whose only purpose is to pass the callback object that should be used.

6.7.2 Stateless Callbacks

A stateless callback interface is a callback whose interface is not marked as *conversational*. Unlike stateful services, a client that uses stateless callbacks will not have callback methods routed to an instance of the client that contains any state that is relevant to the conversation. As such, it is the responsibility of such a client to perform any persistent state management itself. The only information that the client has to work with (other than the parameters of the callback method) is a callback ID object that is passed with requests to the service and is guaranteed to be returned with any callback.

The following is a repeat of the client code fragment above, but with the assumption that in this case the MyServiceCallback is stateless. The client in this case needs to set the callback ID before invoking the service and then needs to get the callback ID when the response is received.

```
534
535         private ServiceReference<MyService> myService;
536
537         @Reference
538         public void setMyService(ServiceReference<MyService> service) {
539             myService = service;
540         }
541
542         public void aClientMethod() {
543             String someKey = "1234";
544             ...
545
546             myService.setCallbackID(someKey);
547             myService.getService().someMethod(arg);
548         }
549
550         @Context RequestContext context;
551
552         public void receiveResult(String result) {
```

```
553         Object key = context.getServiceReference().getCallbackID();
554         // Lookup any relevant state based on "key"
555         // code to process the result
556     }
557
```

558 Just as with stateful callbacks, a service implementation gets access to the callback object by
559 annotating a field or setter method with the @Callback annotation, such as the following:

```
560
561     @Callback
562     protected MyServiceCallback callback;
563
```

564 The difference for stateless services is that the callback field would not be available if the
565 component is servicing a request for anything other than the original client. So, the technique
566 used in the previous section, where there was a response from the backendService which was
567 forwarded as a callback from MyService would not work because the callback field would be null
568 when the message from the backend system was received.

6.7.3 Implementing Multiple Bidirectional Interfaces

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

```
576
577     @Callback
578     protected MyService1Callback callback1;
579
580     @Callback
581     protected MyService2Callback callback2;
582
```

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

6.7.4 Accessing Callbacks

586 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to
587 a Callback instance by annotating a field or method with the **@Callback** annotation.
588
589 A reference implementing the callback service interface may be obtained using
590 CallableReference.getService().

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

```
592
593     @Callback
594     protected CallableReference<MyCallback> callback;
595
596     public void someMethod() {
597
598         MyCallback myCallback = callback.getCallback();     …
599
600         myCallback.receiveResult(theResult);
601     }
602
```

603 Alternatively, a callback may be retrieved programmatically using the **RequestContext** API. The
604 snippet below shows how to retrieve a callback in a method programmatically:

```
606 public void someMethod() {

608     MyCallback myCallback =
609         ComponentContext.getRequestContext().getCallback();

611         …

613     myCallback.receiveResult(theResult);
614 }
```

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

```
619 @Context
620 protected RequestContext requestContext;

622 void receiveResult(Object theResult) {

624     Object refParams =
625         requestContext.getServiceReference().getCallbackID();
626         …
627 }
```

629 On the client side, the object returned by the `getServiceReference()` method represents the
630 service reference for the callback. The object returned by `getCallbackID()` represents the
631 identity associated with the callback, which may be a single String or may be an object (as
632 described below in "Customizing the Callback Identity").

6.7.5 Customizing the Callback

634 By default, the client component of a service is assumed to be the callback service for the
635 bidirectional service. However, it is possible to change the callback by using the
636 **ServiceReference.setCallback()** method. The object passed as the callback should implement
637 the interface defined for the callback, including any additional SCA semantics on that interface
638 such as whether or not it is remotable.

639 Since a service other than the client can be used as the callback implementation, SCA does not
640 generate a deployment-time error if a client does not implement the callback interface of one of its
641 references. However, if a call is made on such a reference without the `setCallback()` method
642 having been called, then a **NoRegisteredCallbackException** is thrown on the client.

643 A callback object for a stateful callback interface has the additional requirement that it must be
644 serializable. The SCA runtime may serialize a callback object and persistently store it.

645 A callback object may be a service reference to another service. In that case, the callback
646 messages go directly to the service that has been set as the callback. If the callback object is not
647 a service reference, then callback messages go to the client and are then routed to the specific
648 instance that has been registered as the callback object. However, if the callback interface has a
649 stateless scope, then the callback object **must** be a service reference.

6.7.6 Customizing the Callback Identity

651 The identity that is used to identify a callback request is initially generated by the system.
652 However, it is possible to provide an application specified identity to identify the callback by calling

653 the **ServiceReference.setCallbackID()** method. This can be used both for stateful and for
654 stateless callbacks. The identity is sent to the service provider, and the binding must guarantee
655 that the service provider will send the ID back when any callback method is invoked.

656 The callback identity has the same restrictions as the conversation ID. It should either be a string
657 or an object that can be serialized into XML. Bindings determine the particular mechanisms to use
658 for transmission of the identity and these may lead to further restrictions when using a given
659 binding.

6.7.7 Bindings for Conversations and Callbacks

661 There are potentially many ways of representing the conversation ID for conversational services
662 depending on the type of binding that is used. For example, it may be possible WS-RM sequence
663 ids for the conversation ID if reliable messaging is used in a Web services binding. WS-Eventing
664 uses a different technique (the wse:Identity header). There is also a WS-Context OASIS TC that
665 is creating a general purpose mechanism for exactly this purpose.

666 SCA's programming model supports conversations, but it leaves up to the binding the means by
667 which the conversation ID is represented on the wire.

7 Java API

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

7.1 Component Context

The following Java code defines the *ComponentContext* interface:

```java
package org.osoaoasisopen.sca;

public interface ComponentContext {

    String getURI();

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

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

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

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

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

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

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

    RequestContext getRequestContext();


}
```

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

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

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

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

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

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

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

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

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

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

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

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

```java
private ComponentContext componentContext;

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

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

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

7.2 Request Context

The following shows the **RequestContext** interface:

```java
package org.osoaoasisopen.sca;

import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();
```

```
766        String getServiceName();
767        <CB> CallableReference<CB> getCallbackReference();
768        <CB> CB getCallback();
769        <B> CallableReference<B> getServiceReference();
770
771    }
772
```

773   The RequestContext interface has the following methods:

774   • **getSecuritySubject()** – Returns the JAAS Subject of the current request

775   • **getServiceName()** – Returns the name of the service on the Java implementation the
776   request came in on

777   • **getCallbackReference()** – Returns a callable reference to the callback as specified by the
778   caller. This method returns null when called for a service request whose interface is not
779   bidirectional or when called for a callback request.

780   • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
781   getCallbackReference() method, this method returns null when called for a service request
782   whose interface is not bidirectional or when called for a callback request.

783   • **getServiceReference()** – When invoked during the execution of a service operation, this
784   method MUST return a CallableReference that represents the service that was invoked.
785   When invoked during the execution of a callback operation, this method MUST return a
786   CallableReference that represents the callback that was invoked.

7.3 CallableReference

788   The following Java code defines the **CallableReference** interface:

789

```
790    package org.osoaoasisopen.sca;
791
792    public interface CallableReference<B> extends java.io.Serializable {
793
794        B getService();
795        Class<B> getBusinessInterface();
796        boolean isConversational();
797        Conversation getConversation();
798        Object getCallbackID();
799    }
800
```

801   The CallableReference interface has the following methods:

802

803   • **getService()** - Returns a type-safe reference to the target of this reference. The instance
804   returned is guaranteed to implement the business interface for this reference. The value
805   returned is a proxy to the target that implements the business interface associated with this
806   reference.

807   • **getBusinessInterface()** – Returns the Java class for the business interface associated with
808   this reference.

809   • **isConversational()** – Returns true if this reference is conversational.

810   • **getConversation()** – Returns the conversation associated with this reference. Returns null if
811   no conversation is currently active.

812   • **getCallbackID()** – Returns the callback ID.

7.4 ServiceReference

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

The following Java code defines the ServiceReference interface:

```
package org.osoaoasisopen.sca;

public interface ServiceReference<B> extends CallableReference<B> {

    Object getConversationID();
    void setConversationID(Object conversationId) throws
        IllegalStateException;
    void setCallbackID(Object callbackID);
    Object getCallback();
    void setCallback(Object callback);
}
```

The ServiceReference interface has the methods of CallableReference plus the following:

- ***getConversationID()*** - Returns the id supplied by the user that will be associated with future conversations initiated through this reference, or null if no ID has been set by the user.
- **setConversationID(*Object conversationId*)** – Set the ID, supplied by the user, to associate with any future conversation started through this reference. If the value supplied is null then the id will be generated by the implementation. Throws an IllegalStateException if a conversation is currently associated with this reference.
- **setCallbackID(*Object callbackID*)** – Sets the callback ID.
- **getCallback()** – Returns the callback object.
- **setCallback(*Object callback*)** – Sets the callback object.

7.5 Conversation

The following snippet defines Conversation:

```
package org.osoaoasisopen.sca;

public interface Conversation {
    Object getConversationID();
    void end();
}
```

The Conversation interface has the following methods:

- **getConversationID()** – Returns the identifier for this conversation. If a user-defined identity had been supplied for this reference then its value will be returned; otherwise the identity generated by the system when the conversation was initiated will be returned.
- **end()** – Ends this conversation.

7.6 ServiceRuntimeException

The following snippet shows the **ServiceRuntimeException**.

```
package org.oasisopen.sca;

public class ServiceRuntimeException extends RuntimeException {
    …
}
```

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

7.7 NoRegisteredCallbackException

The following snippet shows the **NoRegisteredCallbackException**.

```
package org.oasisopen.sca;

public class NoRegisteredCallbackException extends
        ServiceRuntimeException {
    …
}
```
This exception signals a problem where an attempt is made to invoke a callback when a client does not implement the Callback interface and no valid custom Callback has been specified via a call to **ServiceReference.setCallback().**

7.8 ServiceUnavailableException

The following snippet shows the **ServiceUnavailableException**.

```
package org.oasisopen.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    …
}
```

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

7.9 InvalidServiceException

The following snippet shows the **InvalidServiceException**.

```
package org.oasisopen.sca;

public class InvalidServiceException extends ServiceRuntimeException {
    …
}
```

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

7.10 ConversationEndedException

The following snippet shows the **ConversationEndedException**.

```
package org.~~osoa~~oasisopen.sca;

public class ConversationEndedException extends ServiceRuntimeException {
    …
}
```

8 Java Annotations

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

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

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.

8.1 @AllowsPassByReference

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

```
package org.osoaoasisopen.sca.annotations;

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 {

}
```

Formatted: English (U.S.)

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

@AllowsPassByReference has no attributes

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

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

8.2 @ Authentication

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

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

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

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

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

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

The *@Authentication* annotation is used to indicate that the invocation requires authentication. Please check **10.3 Application of Intent Annotations** for samples and details.

8.28.3 @Callback

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

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

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

@Target(TYPE, METHOD, FIELD)
@Retention(RUNTIME)
```

```
1015      public @interface Callback {
1016
1017          Class<?> value() default Void.class;
1018      }
1019
1020
1021  The @Callback annotation is used to annotate a service interface with a callback interface, which
1022  takes the Java Class object of the callback interface as a parameter.
1023  The @Callback annotation has the following attribute:
1024      •   value – the name of a Java class file containing the callback interface
1025
1026  The @Callback annotation may also be used to annotate a method or a field of an SCA
1027  implementation class, in order to have a callback object injected
1028
1029  The following snippet shows a @Callback annotation on an interface:
1030
1031      @Remotable
1032      @Callback(MyServiceCallback.class)
1033      public interface MyService {
1034
1035          void someAsyncMethod(String arg);
1036      }
1037
1038  An example use of the @Callback annotation to declare a callback interface follows:
1039
1040      package somepackage;
1041      import org.osoaoasisopen.sca.annotations.Callback;
1042      import org.osoaoasisopen.sca.annotations.Remotable;
1043      @Remotable
1044      @Callback(MyServiceCallback.class)
1045      public interface MyService {
1046
1047          void someMethod(String arg);
1048      }
1049
1050      @Remotable
1051      public interface MyServiceCallback {
1052
1053          void receiveResult(String result);
1054      }
1055
1056  In this example, the implied component type is:
1057
1058      <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1059
1060          <service name="MyService">
1061              <interface.java interface="somepackage.MyService"
1062                      callbackInterface="somepackage.MyServiceCallback"/>
1063          </service>
1064      </componentType>
```

8.~~3~~8.4 @ComponentName

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

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

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

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

}
```

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

The following snippet shows a component name field definition sample.

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

The following snippet shows a component name setter method sample.

```java
@ComponentName
public void setComponentName(String name) {
 //…
}
```

8.5 @Confidentiality

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

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

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

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

@Inherited
```

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

    /**
     * List of confidentiality qualifiers (such as "message" or "transport").
     *
     * @return confidentiality qualifiers
     */
    @Qualifier
    String[] value() default "";
}
```
The *@Confidentiality* annotation is used to indicate that the invocation requires confidentiality.
Please check **10.3 Application of Intent Annotations** for samples and details.

8.48.6 @Constructor

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

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

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

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

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

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

```
    public class HelloServiceImpl implements HelloService {

      public HelloServiceImpl(){
       ...
       }

       @Constructor
       public HelloServiceImpl(@Property(name="someProperty") String
    someProperty ){
       ...
       }

       public String hello(String message) {
            ...
```

```
1168          }
1169       }
```

8.58.7 @Context

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

```
package org.osoaoasisopen.sca.annotations;

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

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

}
```

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

The @Context annotation has no attributes.

The following snippet shows a ComponentContext field definition sample.

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

The following snippet shows a RequestContext field definition sample.

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

8.68.8 @Conversational

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

```
package org.osoaoasisopen.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
@Target(TYPE)
@Retention(RUNTIME)
public @interface Conversational {
}
```

1216 The @Conversational annotation is used on a Java interface to denote a conversational service
1217 contract.

1218 The @Conversational annotation has no attributes.

1219 The following snippet shows a sample for the @Conversational annotation.

```
1220    package services.hello;
1221
1222    import org.~~osoa~~oasisopen.sca.annotations.Conversational;
1223
1224    @Conversational
1225    public interface HelloService {
1226        void setName(String name);
1227        String sayHello();
1228    }
```

8.78.9 @ConversationAttributes

1229

Formatted: Bullets and Numbering

1230 The following Java code defines the *@ConversationAttributes* annotation:

1231

```
1232    package org.~~osoa~~oasisopen.sca.annotations;
1233
1234    import static java.lang.annotation.ElementType.TYPE;
1235    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1236    import java.lang.annotation.Retention;
1237    import java.lang.annotation.Target;
1238
1239    @Target(TYPE)
1240    @Retention(RUNTIME)
1241    public @interface ConversationAttributes {
1242
1243        String maxIdleTime() default "";
1244        String maxAge() default "";
1245        boolean singlePrincipal() default false;
1246    }
1247
```

Formatted: French (France)

1248 The @ConversationAttributes annotation is used to define a set of attributes which apply to
1249 conversational interfaces of services or references of a Java class. The annotation has the following
1250 attributes:

- 1251 • **maxIdleTime (optional)** - The maximum time that can pass between successive
  1252 operations within a single conversation. If more time than this passes, then the container
  1253 may end the conversation.

- 1254 • **maxAge (optional)** - The maximum time that the entire conversation can remain active.
  1255 If more time than this passes, then the container may end the conversation.

- 1256 • **singlePrincipal (optional)** – If true, only the principal (the user) that started the
  1257 conversation has authority to continue the conversation. The default value is false.

1258

1259 The two attributes that take a time express the time as a string that starts with an integer, is
1260 followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or
1261 "years".

1262

1263 Not specifying timeouts means that timeouts are defined by the SCA runtime implementation,
1264 however it chooses to do so.

1265

1266 The following snippet shows the use of the @ConversationAttributes annotation to set the
1267 maximum age for a Conversation to be 30 days.

1268

```
1269    package service.shoppingcart;
1270
1271    import org.~~osoa~~oasisopen.sca.annotations.ConversationAttributes;
1272
1273    @ConversationAttributes (maxAge="30 days");
1274    public class ShoppingCartServiceImpl implements ShoppingCartService {
1275        ...
1276    }
```

1277 ## ~~8.8~~8.10 @ConversationID

1278 The following Java code defines the *@ConversationID* annotation:

1279

```
1280    package org.~~osoa~~oasisopen.sca.annotations;
1281
1282    import static java.lang.annotation.ElementType.METHOD;
1283    import static java.lang.annotation.ElementType.FIELD;
1284    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1285    import java.lang.annotation.Retention;
1286    import java.lang.annotation.Target;
1287
1288    @Target({METHOD, FIELD})
1289    @Retention(RUNTIME)
1290    public @interface ConversationID {
1291
1292    }
1293
```

1294 The @ConversationID annotation is used to annotate a Java class field or setter method that is
1295 used to inject the conversation ID. System generated conversation IDs are always strings, but
1296 application generated conversation IDs may be other complex types.

1297 The following snippet shows a conversation ID field definition sample.

1298

```
1299    @ConversationID
1300    private String conversationID;
```

1301

1302 The type of the field is not necessarily String.

1303

1304 ## ~~8.9~~8.11 @Destroy

1305 The following Java code defines the *@Destroy* annotation:

1306

```
1307    package org.~~osoa~~oasisopen.sca.annotations;
1308
1309    import static java.lang.annotation.ElementType.METHOD;
1310    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1311    import java.lang.annotation.Retention;
```

```
1312    import java.lang.annotation.Target;
1313
1314    @Target(METHOD)
1315    @Retention(RUNTIME)
1316    public @interface Destroy {
1317
1318    }
1319
```

1320   The @Destroy annotation is used to denote a single Java class method that will be called when the
1321   scope defined for the implementation class ends. The method MAY have any access modifier and
1322   MUST have a void return type and no arguments.

1323   If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1324   when the scope defined for the implementation class ends. If the implementation class has a
1325   method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1326   NOT instantiate the implementation class.

1328   The following snippet shows a sample for a destroy method definition.

```
1330    @Destroy
1331    public void myDestroyMethod() {
1332        …
1333    }
```

8.108.12 @EagerInit

1334

1335   The following Java code defines the @*EagerInit* annotation:

```
1337    package org.osoaoasisopen.sca.annotations;
1338
1339    import static java.lang.annotation.ElementType.TYPE;
1340    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1341    import java.lang.annotation.Retention;
1342    import java.lang.annotation.Target;
1343
1344    @Target(TYPE)
1345    @Retention(RUNTIME)
1346    public @interface EagerInit {
1347
1348    }
1349
```

1350   The @*EagerInit* annotation is used to annotate the Java class of a COMPOSITE scoped
1351   implementation for eager initialization. When marked for eager initialization, the composite scoped
1352   instance is created when its containing component is started.

8.118.13 @EndsConversation

1353

1354   The following Java code defines the @*EndsConversation* annotation:

```
1356    package org.osoaoasisopen.sca.annotations;
1357
1358    import static java.lang.annotation.ElementType.METHOD;
1359    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1360    import java.lang.annotation.Retention;
```

```
1361        import java.lang.annotation.Target;
1362
1363        @Target(METHOD)
1364        @Retention(RUNTIME)
1365        public @interface EndsConversation {
1366
1367
1368        }
1369
```

1370   The @EndsConversation annotation is used to denote a method on a Java interface that is called
1371   to end a conversation.

1372   The @EndsConversation annotation has no attributes.

1373   The following snippet shows a sample using the @EndsConversation annotation.

```
1374        package services.shoppingbasket;
1375
1376        import org.osoaoasisopen.sca.annotations.EndsConversation;
1377
1378        public interface ShoppingBasket {
1379            void addItem(String itemID, int quantity);
1380
1381            @EndsConversation
1382            void buy();
1383        }
```

1384   8.128.14 @Init

1385   The following Java code defines the **@Init** annotation:

1386

```
1387        package org.osoaoasisopen.sca.annotations;
1388
1389        import static java.lang.annotation.ElementType.METHOD;
1390        import static java.lang.annotation.RetentionPolicy.RUNTIME;
1391        import java.lang.annotation.Retention;
1392        import java.lang.annotation.Target;
1393
1394        @Target(METHOD)
1395        @Retention(RUNTIME)
1396        public @interface Init {
1397
1398
1399        }
1400
```

1401   The @Init annotation is used to denote a single Java class method that is called when the scope
1402   defined for the implementation class starts. The method MAY have any access modifier and MUST
1403   have a void return type and no arguments.

1404   If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1405   after all property and reference injection is complete. If the implementation class has a method
1406   with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1407   instantiate the implementation class.

1408   The following snippet shows an example of an init method definition.

1409

```
1410        @Init
1411        public void myInitMethod() {
```

```
1412            …
1413        }
1414
```

8.15 @Integrity

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

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

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

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

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

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

The **@Integrity** annotation is used to indicate that the invocation requires integrity. Please check **10.3 Application of Intent Annotations** for samples and details.

8.138.16 @OneWay

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

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

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

```
1464    @Target(METHOD)
1465    @Retention(RUNTIME)
1466    public @interface OneWay {
1467
1468
1469        }
1470
```

1471   The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1472   will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1473   Programming.

1474   The @OneWay annotation has no attributes.

1475   The following snippet shows the use of the @OneWay annotation on an interface.

```
1476    package services.hello;
1477
1478    import org.osoaoasisopen.sca.annotations.OneWay;
1479
1480    public interface HelloService {
1481        @OneWay
1482        void hello(String name);
1483    }
1484
```

8.17 @PolicySets

1486
1487   The following Java code defines the *@PolicySets* annotation:

```
1488
1489    package org.oasisopen.sca.annotations;
1490
1491    import static java.lang.annotation.ElementType.FIELD;
1492    import static java.lang.annotation.ElementType.METHOD;
1493    import static java.lang.annotation.ElementType.PARAMETER;
1494    import static java.lang.annotation.ElementType.TYPE;
1495    import static java.lang.annotation.RetentionPolicy.RUNTIME;
1496
1497    import java.lang.annotation.Retention;
1498    import java.lang.annotation.Target;
1499
1500    @Target({TYPE, FIELD, METHOD, PARAMETER})
1501    @Retention(RUNTIME)
1502    public @interface PolicySets {
1503        /**
1504         * Returns the policy sets to be applied.
1505         *
1506         * @return the policy sets to be applied
1507         */
1508        String[] value() default "";
1509    }
1510
```

1511   The *@PolicySet* annotation is used to describe SCA Policy Sets. Please check **10.5 Policy Set**
1512   **Annotations** for samples and details.

1513
1514

Formatted: Font color: Custom Color(RGB(59,0,111))

Formatted: Heading 2,H2, Indent: Left: 0", First line: 0", Automatically adjust right indent when grid is defined, Adjust space between Latin and Asian text, Adjust space between Asian text and numbers

Formatted: English (U.S.)

8.148.18 @Property

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

```
package org.osoaoasisopen.sca.annotations;

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

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

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

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

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

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

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

The @Property annotation has the following attributes:

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

- **required (optional)** – specifies whether injection is required, defaults to true. For a constructor parameter annotation, this attribute MUST have the value true.

The following snippet shows a property field definition sample.

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

The following snippet shows a property setter sample

```
1564
1565        @Property(name="currency", required=true)
1566        public void setCurrency( String theCurrency ) {

1567            ....

1568        }

1569

1570    If the property is defined as an array or as any type that extends or implements
1571    *java.util.Collection*, then the implied component type has a property with a *many* attribute set to
1572    true.

1573

1574    The following snippet shows the definition of a configuration property using the @Property
1575    annotation for a collection.

1576

1577        ...
1578        private List<String> helloConfigurationProperty;

1579
1580        @Property(required=true)
1581        public void setHelloConfigurationProperty(List<String> property) {
1582                helloConfigurationProperty = property;
1583        }
1584        ...
1585
```

8.19 @Qualifier

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

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

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

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

/**
 * Annotation that can be applied to an attribute of an @Intent annotation to
indicate the
 * attribute provides qualifiers for the intent.
 *
 * @version $Rev: 537744 $ $Date: 2007-05-14 04:00:18 -0400 (Mon, 14 May
2007) $
 */
@Target(METHOD)
@Retention(RUNTIME)
public @interface Qualifier {
}
```

The *@Qualifier* annotation can be applied to an attribute as an @Intent annotation to indicate the

attribute prodies qualifiers for the intent. Please check **10.3 Application of Intent**

Annotations for samples and details.

8.158.20 @Reference

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

```
package org.osoaoasisopen.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Reference {

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

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

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

References may 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.

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

The @Reference annotation has the following attributes:

- **name (optional)** – the name of the reference. For a field annotation, the default is the name of the field of the Java class. For a setter method annotation, the default is the JavaBeans property name corresponding to the setter method name. For a constructor parameter annotation, there is no default and the name attribute MUST be present.

- **required (optional)** – whether injection of service or services is required. Defaults to true. For a constructor parameter annotation, this attribute MUST have the value true.

The following snippet shows a reference field definition sample.

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

The following snippet shows a reference setter sample

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

```
1662           ...
1663       }
1664
```

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

```
1670       package services.hello;
1671
1672       private HelloService helloService;
1673
1674       @Reference(name="helloService", required=true)
1675       public setHelloService(HelloService service) {
1676               helloService = service;
1677       }
1678
1679       public void clientMethod() {
1680               String result = helloService.hello("Hello World!");
1681               …
1682       }
1683
```

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

```
1688       <?xml version="1.0" encoding="ASCII"?>
1689       <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1690
1691           <!-- Any services offered by the component would be listed here -->
1692           <reference name="helloService" multiplicity="1..1">
1693               <interface.java interface="services.hello.HelloService"/>
1694           </reference>
1695
1696       </componentType>
1697
```

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

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

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

```
1712
1713           @Reference(name="helloServices", required=true)
```

```
1714        protected List<HelloService> helloServices;
1715
1716        public void clientMethod() {
1717
1718            …
1719            for (int index = 0; index < helloServices.size(); index++) {
1720                HelloService helloService =
1721                    (HelloService)helloServices.get(index);
1722                String result = helloService.hello("Hello World!");
1723            }
1724            …
1725        }
1726
```

1727 The following snippet shows the XML representation of the component type reflected from for the
1728 former component implementation fragment. There is no need to author this component type in
1729 this case since it can be reflected from the Java class.

1730

```
1731    <?xml version="1.0" encoding="ASCII"?>
1732    <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1733
1734        <!-- Any services offered by the component would be listed here -->
1735        <reference name="helloServices" multiplicity="1..n">
1736            <interface.java interface="services.hello.HelloService"/>
1737        </reference>
1738
1739    </componentType>
```

1740

1741 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1742 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1743 of 0..N must be an empty array or collection.

8.15.18.20.1 Reinjection

1744

1745 References MAY be reinjected after the initial creation of a component if the reference target
1746 changes due to a change in wiring that has occurred since the component was initialized. In order
1747 for reinjection to occur, the following MUST be true:

1748   1. The component MUST NOT be STATELESS scoped.

1749   2. The reference MUST use either field-based injection or setter injection. References that are
1750   injected through constructor injection MUST NOT be changed. Setter injection allows for
1751   code in the setter method to perform processing in reaction to a change.

1752   3. If the reference has a conversational interface, then reinjection MUST NOT occur while the
1753   conversation is active.

1754 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1755 work as if the reference target was not changed.

1756 If an operation is called on a reference where the target of that reference has been undeployed,
1757 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
1758 where the target of the reference has become unavailable for some reason, the SCA runtime
1759 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
1760 reference MAY continue to work, depending on the runtime and the type of change that was made.
1761 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1762 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
1763 corresponds to the reference that is passed as a parameter to cast(). If the reference is
1764 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
1765 to work as if the reference target was not changed. If the target of a ServiceReference has been

1766 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
1767 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
1768 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
1769 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
1770 work, depending on the runtime and the type of change that was made. If it doesn't work, the
1771 exception thrown will depend on the runtime and the cause of the failure.

1772 A reference or ServiceReference accessed through the component context by calling getService()
1773 or getServiceReference() MUST correspond to the current configuration of the domain. This
1774 applies whether or not reinjection has taken place. If the target has been undeployed or has
1775 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
1776 and attempts to call business methods SHOULD throw an exception as described above. If the
1777 target has changed, the result SHOULD be a reference to the changed service.

1778 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
1779 means that in the cases listed above where reference reinjection is not allowed, the array or
1780 Collection for the reference MUST NOT change its contents. In cases where the contents of a
1781 reference collection MAY change, then for references that use setter injection, the setter method
1782 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
1783 the same array or Collection object previously injected to the component.

1784

	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 undeployed	Business methods SHOULD throw InvalidServiceException.	Business methods SHOULD throw InvalidServiceException.	Result SHOULD be a reference to the undeployed 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.

* Other conditions:

1. The component MUST NOT be STATELESS scoped.

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

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

1785

8.168.21 @Remotable

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

```
package org.osoaoasisopen.sca.annotations;

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


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

}
```

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

The @Remotable annotation has no attributes.

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

```
package services.hello;

import org.osoaoasisopen.sca.annotations.*;

@Remotable
public interface HelloService {

    String hello(String message);
}
```

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

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

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

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

The following snippet shows a remotable Java service interface.

```
package services.hello;

import org.~~osoa~~oasisopen.sca.annotations.*;

@Remotable
public interface HelloService {

    String hello(String message);
}

package services.hello;

import org.~~osoa~~oasisopen.sca.annotations.*;

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

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

8.22 @Requires

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

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

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

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

@Inherited
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER}))
public @interface Requires {
    /**
     * Returns the attached intents.
     *
     * @return the attached intents
     */
    String[] value() default "";
}
```

The *@Requires* annotation supports general purpose intents specified as strings. User may also define specific intents using @Intent annotation. Please check **10.1 General Intent Annotations** for samples and details.

8.178.23 @Scope

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

```
package org.osoaoasisopen.sca.annotations;

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

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

    String value() default "STATELESS";
}
```

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

The @Scope annotation has the following attribute:

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

The default value is STATELESS, except for an implementation offering a @Conversational service, which has a default scope of CONVERSATION. See section 2.2 for more details of the SCA-defined scopes.

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

```
package services.hello;

import org.osoaoasisopen.sca.annotations.*;

@Service(HelloService.class)
@Scope("CONVERSATION")
public class HelloServiceImpl implements HelloService {

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

8.188.24 @Service

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

```
package org.osoaoasisopen.sca.annotations;

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

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

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

    Class<?>[] interfaces() default {};
    Class<?> value() default Void.class;
}
```

Formatted: French (France)

Formatted: English (U.S.)

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

The @Service annotation has the following attributes:

- *interfaces* – The value is an array of interface or class objects that should be exposed as services by this component.

- *value* – A shortcut for the case when the class provides only a single service interface.

Only one of these attributes should be specified.

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

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

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

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

```
package services.hello;

import org.oasisopen.sca.annotations.Service;

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

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

8.25 Security Implementation Policy Annotations

JSR 250 "Common Annotations for the Java Platform" defines the following annotations that can be used for implementation security policy:

1992
1993　javax.annotation.security.RunAs
1994　javax.annotation.security.RolesAllowed
1995　javax.annotation.security.PermitAll
1996　javax.annotation.security.DenyAll
1997　javax.annotation.security.DeclareRoles
1998
1999　Based on JSR 250, the RunAs , DeclareRoles annotations can be specified on a class; the RolesAllowed
2000　, PermitAll , DenyAll annotations can be specified on a class or on method(s).
2001
2002　Please check JSR250 and SCA Policy spec for details on the meaning of the annotations . Please check
2003　the section [10.6.2] Security Implementation Policy for the details on how these annotations are
2004　mapped into Policy framework.
2005

Formatted: Font: (Default) Verdana, 9 pt

Formatted: Font: (Default) Verdana, 9 pt

9 WSDL to Java and Java to WSDL

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

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

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

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

- No support for holders

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

9.1 JAX-WS Client Asynchronous API for a Synchronous Service

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

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

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

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

    b. the same parameter signature as M, and

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

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

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

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

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

    c. a return type of Future<?>

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

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

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

```
2050      <part name="ticker" type="xsd:string"/>
2051     </message>
2052
2053     <message name="getPriceResponse">
2054      <part name="price" type="xsd:float"/>
2055     </message>
2056
2057     <portType name="StockQuote">
2058      <operation name="getPrice">
2059         <input message="tns:getPrice"/>
2060         <output message="tns:getPriceResponse"/>
2061      </operation>
2062     </portType>
```

2063

2064   The JAX-WS asynchronous mapping will produce the following Java interface:

```
2065     // asynchronous mapping
2066     @WebService
2067     public interface StockQuote {
2068      float getPrice(String ticker);
2069      Response<Float> getPriceAsync(String ticker);
2070      Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2071     }
```

2072

2073   For SCA interface definition purposes, this is treated as equivalent to the following:

```
2074     // synchronous mapping
2075     @WebService
2076     public interface StockQuote {
2077      float getPrice(String ticker);
2078     }
```

2079

2080   SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2081   example, if the client implementation uses the asynchronous form of the interface, the two
2082   additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2083   WS specification.

10 Policy Annotations for Java

2084

2085 SCA provides facilities for the attachment of policy-related metadata to SCA assemblies, which
2086 influence how implementations, services and references behave at runtime. The policy facilities
2087 are described in the SCA Policy Framework specification [POLICY]. In particular, the facilities
2088 include Intents and Policy Sets, where intents express abstract, high-level policy requirements and
2089 policy sets express low-level detailed concrete policies.

2090 Policy metadata can be added to SCA assemblies through the means of declarative statements
2091 placed into Composite documents and into Component Type documents. These annotations are
2092 completely independent of implementation code, allowing policy to be applied during the assembly
2093 and deployment phases of application development.

2094 However, it can be useful and more natural to attach policy metadata directly to the code of
2095 implementations. This is particularly important where the policies concerned are relied on by the
2096 code itself. An example of this from the Security domain is where the implementation code
2097 expects to run under a specific security Role and where any service operations invoked on the
2098 implementation must be authorized to ensure that the client has the correct rights to use the
2099 operations concerned. By annotating the code with appropriate policy metadata, the developer
2100 can rest assured that this metadata is not lost or forgotten during the assembly and deployment
2101 phases.

2102 The SCA Java Common Annotations specification provides a series of annotations which provide
2103 the capability for the developer to attach policy information to Java implementation code. The
2104 annotations concerned first provide general facilities for attaching SCA Intents and Policy Sets to
2105 Java code. Secondly, there are further specific annotations that deal with particular policy intents
2106 for certain policy domains such as Security.

2107 The SCA Java Common Annotations specification supports using the Common Annotation for Java
2108 Platform specification (JSR-250) [JSR-250]. An implication of adopting the common annotation
2109 for Java platform specification is that the SCA Java specification support consistent annotation and
2110 Java class inheritance relationships.

2111

10.1 General Intent Annotations

2113 SCA provides the annotation *@Requires* for the attachment of any intent to a Java class, to a
2114 Java interface or to elements within classes and interfaces such as methods and fields.

2115 The @Requires annotation can attach one or multiple intents in a single statement.

2116 Each intent is expressed as a string. Intents are XML QNames, which consist of a Namespace URI
2117 followed by the name of the Intent. The precise form used follows the string representation used
2118 by the javax.xml.namespace.QName class, which is as follows:

2119   "{" + *Namespace URI* + "}" + *intentname*

2120 Intents may be qualified, in which case the string consists of the base intent name, followed by a
2121 ".", followed by the name of the qualifier. There may also be multiple levels of qualification.

2122 This representation is quite verbose, so we expect that reusable String constants will be defined
2123 for the namespace part of this string, as well as for each intent that is used by Java code. SCA
2124 defines constants for intents such as the following:

2125   public static final String SCA_PREFIX="{http://docs.oasis-
2126 open.org/ns/opencsa/sca/200712}";

2127   public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";

2128   public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";

2129 Notice that, by convention, qualified intents include the qualifier as part of the name of the
2130 constant, separated by an underscore. These intent constants are defined in the file that defines
2131 an annotation for the intent (annotations for intents, and the formal definition of these constants,
2132 are covered in a following section).

2133 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

2134 An example of the @Requires annotation with 2 qualified intents (from the Security domain)
2135 follows:

2136

2137 ```
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2138

2139 This attaches the intents "confidentiality.message" and "integrity.message".

2140 The following is an example of a reference requiring support for confidentiality:

2141 ```
package
2142 org.osoaoasisopen.sca.annotation;org.oasisopen.sca.annotations;
2143

2144 import static org.osoaoasisopen.sca.annotations.Confidentiality.*;
2145

2146 public class Foo {
2147     @Requires(CONFIDENTIALITY)
2148     @Reference
2149     public void setBar(Bar bar) {
2150         …
2151     }
2152 }
```

2153 Users may also choose to only use constants for the namespace part of the QName, so that they
2154 may add new intents without having to define new constants. In that case, this definition would
2155 instead look like this:

2156 ```
package
2157 org.osoaoasisopen.sca.annotation;org.oasisopen.sca.annotations;
2158

2159 import static org.osoaoasisopen.sca.Constants.*;
2160

2161 public class Foo {
2162   @Requires(SCA_PREFIX+"confidentiality")
2163   @Reference
2164   public void setBar(Bar bar) {
2165   …
2166   }
2167 }
```

2168

2169 The formal syntax for the @Requires annotation follows:

2170    @Requires( "qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]}

2171  where

2172    qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

2173

2174  The following shows the formal definition of the @Requires annotation:

2175

```
2176    package org.osoaoasisopen.sca.annotations;
2177    import static java.lang.annotation.ElementType.TYPE;
2178    import static java.lang.annotation.ElementType.METHOD;
2179    import static java.lang.annotation.ElementType.FIELD;
2180    import static java.lang.annotation.ElementType.PARAMETER;
2181    import static java.lang.annotation.RetentionPolicy.RUNTIME;
2182    import java.lang.annotation.Retention;
2183    import java.lang.annotation.Target;
2184    import java.lang.annotation.Inherited;
2185
2186    @Inherited
2187    @Retention(RUNTIME)
2188    @Target({TYPE, METHOD, FIELD, PARAMETER})
2189
2190    public @interface Requires {
2191        String[] value() default "";
2192    }
```

2193 The SCA_NS constant is defined in the Constants interface:

```
2194    package org.osoaoasisopen.sca;
2195
2196    public interface Constants {
2197        String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
2198        String SCA_PREFIX = "{"+SCA_NS+"}";
2199    }
```

2200

10.2 Specific Intent Annotations

2202 In addition to the general intent annotation supplied by the @Requires annotation described
2203 above, it is also possible to have Java annotations that correspond to specific policy intents. SCA
2204 provides a number of these specific intent annotations and it is also possible to create new specific
2205 intent annotations for any intent.

2206 The general form of these specific intent annotations is an annotation with a name derived from
2207 the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an
2208 attribute to the annotation in the form of a string or an array of strings.

2209 For example, the SCA confidentiality intent described in the section on General Intent Annotations
2210 using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific
2211 @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent
2212 is:

2213   @Integrity

2214 An example of a qualified specific intent for the "authentication" intent is:

2215   @Authentication({"message", "transport"})

2216 This annotation attaches the pair of qualified intents: "authentication.message" and
2217 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
2218 "http://docs.oasis-open.org/ns/opencsa/sca/200712").

2219 The general form of specific intent annotations is:

2220   @<Intent>[(qualifiers)]

2221 where Intent is an NCName that denotes a particular type of intent.

2222   Intent ::= NCName

2223   qualifiers ::= "qualifier" | {"qualifier" [, "qualifier"] }

2224   qualifier ::= NCName | NCName/qualifier

2225

2226 10.2.1 How to Create Specific Intent Annotations

2227 SCA identifies annotations that correspond to intents by providing an @Intent annotation which
2228 must be used in the definition of an intent annotation.

2229 The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
2230 String form of the QName of the intent. As part of the intent definition, it is good practice
2231 (although not required) to also create String constants for the Namespace, the Intent and for
2232 Qualified versions of the Intent (if defined). These String constants are then available for use with
2233 the @Requires annotation and it should also be possible to use one or more of them as
2234 parameters to the @Intent annotation.

2235 Alternatively, the QName of the intent may be specified using separate parameters for the
2236 targetNamespace and the localPart for example:

2237   @Intent(targetNamespace=SCA_NS, localPart="confidentiality").

2238 The definition of the @Intent annotation is the following:

2239

2240   **package**
2241   ~~org.osoaoasisopen.sca.annotation;~~org.oasisopen.sca.annotations;

2242   **import static** java.lang.annotation.ElementType.ANNOTATION_TYPE;

2243   **import static** java.lang.annotation.RetentionPolicy.RUNTIME;

2244   **import** java.lang.annotation.Retention;   [Formatted: English (U.S.)]

2245   **import** java.lang.annotation.Target;

2246   **import** java.lang.annotation.Inherited;

2247

2248   @Retention(RUNTIME)

2249   @Target(ANNOTATION_TYPE)

2250   **public @interface** Intent {

2251   String value() default "";

2252	`        String targetNamespace() default "";`
2253	`        String localPart() default "";`
2254	`    }`

2255 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a
2256 string (or an array of strings) which holds one or more qualifiers.

2257 In this case, the attribute's definition should be marked with the @Qualifier annotation. The
2258 @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent
2259 represented by the whole annotation. If more than one qualifier value is specified in an
2260 annotation, it means that multiple qualified forms are required. For example:

2261 `    @Confidentiality({"message","transport"})`

2262 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
2263 are set for the element to which the confidentiality intent is attached.

2264 The following is the definition of the @Qualifier annotation.

2265

2266 `    package org.osoaoasisopen.sca.annotations;`

2267 `    import static java.lang.annotation.ElementType.METHOD;`

2268 `    import static java.lang.annotation.RetentionPolicy.RUNTIME;`

2269 `    import java.lang.annotation.Retention;`

2270 `    import java.lang.annotation.Target;`

2271 `    import java.lang.annotation.Inherited;`

2272

2273 `    @Retention(RetentionPolicy.RUNTIME)`

2274 `    @Target(ElementType.METHOD)`

2275 `    public @interface Qualifier {`

2276 `    }`

2277

2278 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
2279 intent annotations are shown in the section dealing with Security Interaction Policy.

2280

2281 10.3 Application of Intent Annotations

2282 The SCA Intent annotations can be applied to the following Java elements:

2283 • Java class

2284 • Java interface

2285 • Method

2286 • Field

2287 Where multiple intent annotations (general or specific) are applied to the same Java element, they
2288 are additive in effect. An example of multiple policy annotations being used together follows:

2289 `    @Authentication`
2290 `    @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})`

2291 In this case, the effective intents are "authentication", "confidentiality.message" and
2292 "integrity.message".

Formatted: English (U.S.)

2293 If an annotation is specified at both the class/interface level and the method or field level, then
2294 the method or field level annotation completely overrides the class level annotation of the same
2295 type.

2296 The intent annotation can be applied either to classes or to class methods when adding annotated
2297 policy on SCA services. Applying an intent to the setter method in a reference injection approach
2298 allows intents to be defined at references.

10.3.1 Inheritance And Annotation

2300 The inheritance rules for annotations are consistent with the common annotation specification, JSR
2301 250.

2302 The following example shows the inheritance relations of intents on classes, operations, and super
2303 classes.

2304

```
2305    package services.hello;
2306    import org.osoaoasisopen.sca.annotations.Remotable;
2307    import org.osoaoasisopen.sca.annotations.Integrity;
2308    import org.osoaoasisopen.sca.annotations.Authentication;
2309
2310    @Integrity("transport")
2311    @Authentication
2312    public class HelloService {
2313            @Integrity
2314            @Authentication("message")
2315            public String hello(String message) {...}
2316
2317            @Integrity
2318            @Authentication("transport")
2319            public String helloThere() {...}
2320    }
2321
2322    package services.hello;
2323    import org.osoaoasisopen.sca.annotations.Remotable;
2324    import org.osoaoasisopen.sca.annotations.Confidentiality;
2325    import org.osoaoasisopen.sca.annotations.Authentication;
2326
2327    @Confidentiality("message")
2328    public class HelloChildService extends HelloService {
2329            @Confidentiality("transport")
2330            public String hello(String message) {...}
2331            @Authentication
2332            String helloWorld() {...}
2333    }
```

2334 Example 2a. Usage example of annotated policy and inheritance.

2335

2336 The effective intent annotation on the helloWorld method is Integrity("transport"),
2337 @Authentication, and @Confidentiality("message").

2338 The effective intent annotation on the hello method of the HelloChildService is
2339 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),

2340 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity
2341 and @Authentication("transport"), the same as in HelloService class.

2342 The effective intent annotation on the hello method of the HelloService is @Integrity and
2343 @Authentication("message")

2344

2345 The listing below contains the equivalent declarative security interaction policy of the HelloService
2346 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
2347 Example 2a.

2348

```
2349    <?xml version="1.0" encoding="ASCII"?>
2350
2351    <composite  xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2352                    name="HelloServiceComposite" >
2353        <service name="HelloService" requires="integrity/transport
2354            authentication">
2355            …
2356        </service>
2357        <service name="HelloChildService" requires="integrity/transport
2358            authentication confidentiality/message">
2359            …
2360        </service>
2361        ...
2362
2363        <component name="HelloServiceComponent">*
2364            <implementation.java class="services.hello.HelloService"/>
2365                    <operation name="hello" requires="integrity
2366                        authentication/message"/>
2367                    <operation name="helloThere"
2368    requires="integrity
2369                            authentication/transport"/>
2370        </component>
2371        <component name="HelloChildServiceComponent">*
2372            <implementation.java
2373    class="services.hello.HelloChildService" />
2374            <operation name="hello"
2375    requires="confidentiality/transport"/>
2376            <operation name="helloThere" requires=" integrity/transport
2377                authentication"/>
2378            <operation name=helloWorld" requires="authentication"/>
2379        </component>
2380
2381        ...
2382
2383    </composite>
2384
```

2385 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

2386

10.4 Relationship of Declarative And Annotated Intents

Annotated intents on a Java class cannot be overridden by declarative intents either in a composite document which uses the class as an implementation or by statements in a component Type document associated with the class. This rule follows the general rule for intents that they represent fundamental requirements of an implementation.

An unqualified version of an intent expressed through an annotation in the Java class may be qualified by a declarative intent in a using composite document.

10.5 Policy Set Annotations

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

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

```
@PolicySets( "<policy set QName>" |
             { "<policy set QName>" [, "<policy set QName>"] })
```

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

An example of the @PolicySets annotation:

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

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

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

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

- Java class
- Java interface
- Method
- Field

10.6 Security Policy Annotations

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

10.6.1 Security Interaction Policy

2429

2430 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply
2431 to the operation of services and references of an implementation:

2432 • @Integrity

2433 • @Confidentiality

2434 • @Authentication

2435 All three of these intents have the same pair of Qualifiers:

2436 • message

2437 • transport

2438 The following snippets shows the @Integrity, @Confidentiality and @Authentication annotations:

```
2439    package org.osoaoasisopen.sca.annotations;
2440
2441    import java.lang.annotation.*;
2442    import static org.osoaoasisopen.sca.Constants.SCA_NS;
2443
2444    @Inherited
2445    @Retention(RetentionPolicy.RUNTIME)
2446    @Target({ElementType.TYPE,ElementType.METHOD,
2447                      ElementType.FIELD, ElementType.PARAMETER})
2448    @Intent(Integrity.INTEGRITY)
2449    public @interface Integrity {
2450        String INTEGRITY = SCA_NS+"integrity";
2451        String INTEGRITY_MESSAGE = INTEGRITY+".message";
2452        String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
2453        @Qualifier
2454        String[] value() default "";
2455    }
2456
2457
2458    package org.osoaoasisopen.sca.annotations;
2459
2460    import java.lang.annotation.*;
2461    import static org.osoaoasisopen.sca.Constants.SCA_NS;
2462
2463    @Inherited
2464    @Retention(RetentionPolicy.RUNTIME)
2465    @Target({ElementType.TYPE,ElementType.METHOD,
2466                      ElementType.FIELD, ElementType.PARAMETER})
2467    @Intent(Confidentiality.CONFIDENTIALITY)
```

```
2468          public @interface Confidentiality {
2469              String CONFIDENTIALITY = SCA_NS+"confidentiality";
2470              String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";
2471              String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2472              @Qualifier
2473              String[] value() default "";
2474          }
```

2475

2476

```
2477          package org.osoaoasisopen.sca.annotations;
```

2478

```
2479          import java.lang.annotation.*;
2480          import static org.osoaoasisopen.sca.Constants.SCA_NS;
```

2481

```
2482          @Inherited
2483          @Retention(RetentionPolicy.RUNTIME)
2484          @Target({ElementType.TYPE,ElementType.METHOD,
2485                             ElementType.FIELD, ElementType.PARAMETER})
2486          @Intent(Authentication.AUTHENTICATION)
2487          public @interface Authentication {
2488              String AUTHENTICATION = SCA_NS+"authentication";
2489              String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
2490              String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
2491              @Qualifier
2492              String[] value() default "";
2493          }
```

2494

2495

2496   The following example shows an example of applying an intent to the setter method used to inject
2497   a reference.   Accessing the hello operation of the referenced HelloService requires both
2498   "integrity.message" and "authentication.message" intents to be honored.

2499

```
2500          //Interface for HelloService
2501          public interface service.hello.HelloService {
2502                String hello(String helloMsg);
2503          }
```

2504

```
2505          // Interface for ClientService
2506          public interface service.client.ClientService {
2507                public void clientMethod();
```

```
2508            }

2510        // Implementation class for ClientService
2511        package services.client;

2513        import services.hello.HelloService;

2515        import org.osoaoasisopen.sca.annotations.*;

2517        @Service(ClientService.class)
2518        public class ClientServiceImpl implements ClientService {


2521                private HelloService helloService;

2523                @Reference(name="helloService", required=true)
2524                @Integrity("message")
2525                @Authentication("message")
2526                public void setHelloService(HelloService service) {
2527                        helloService = service;
2528                }

2530                public void clientMethod() {
2531                        String result = helloService.hello("Hello World!");
2532                        …
2533                }
2534        }
```

Example 1. Usage of annotated intents on a reference.

10.6.2 Security Implementation Policy

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

- RunAs

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

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

- RolesAllowed

    Takes as a parameter a single string or an array of strings which represent one or more role names. When present, the implementation can only be accessed by principals whose

2553 role corresponds to one of the role names listed in the @roles attribute. How role names
2554 are mapped to security principals is implementation dependent (SCA does not define this).
2555 eg. @RolesAllowed({"Manager", "Employee"})

- PermitAll

2556
2557
2558 No parameters. When present, grants access to all roles.

- DenyAll

2559
2560
2561 No parameters. When present, denies access to all roles.

- DeclareRoles

2562
2563 Takes as a parameter a string or an array of strings which identify one or more role names
2564 that form the set of roles used by the implementation.
2565 eg. @DeclareRoles({"Manager", "Employee", "Customer"})

2566 (all these are declared in the Java package javax.annotation.security)

2567
2568 None of these annotations will appear in the introspected componentType definitions, therefore, these
2569 policies are Java implementation specific policies which only Java implementation implementers needs
2570 to deal with.
2571

2572 For a full explanation of these intents, see the Policy Framework specification [POLICY].

10.6.2.1 Annotated Implementation Policy Example

2574 The following is an example showing annotated security implementation policy:

2575

```
2576     package services.account;
2577     @Remotable
2578     public interface AccountService {
2579         AccountReport getAccountReport(String customerID);
2580     }
```

2581

2582 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,
2583 plus the service references it makes and the settable properties that it has, along with a set of
2584 implementation policy annotations:

2585

```
2586     package services.account;
2587     import java.util.List;
2588     import commonj.sdo.DataFactory;
2589     import org.osoaoasisopen.sca.annotations.Property;
2590     import org.osoaoasisopen.sca.annotations.Reference;
2591     import
2592     javax.annotation.securityorg.osoaoasisopen.sca.annotations.RolesAllowed;
2593     import javax.annotation.security.org.osoaoasisopen.sca.annotations.RunAs;
2594     import
2595     javax.annotation.securityorg.osoaoasisopen.sca.annotations.PermitAll;
2596     import services.accountdata.AccountDataService;
2597     import services.accountdata.CheckingAccount;
2598     import services.accountdata.SavingsAccount;
2599     import services.accountdata.StockAccount;
```

```java
import services.stockquote.StockQuoteService;
@RolesAllowed("customers")
@RunAs("accountants" )
public class AccountServiceImpl implements AccountService {

    @Property
    protected String currency = "USD";

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

    @RolesAllowed({"customers", "accountants"})
    public AccountReport getAccountReport(String customerID) {

     DataFactory dataFactory = DataFactory.INSTANCE;
     AccountReport accountReport =
          (AccountReport)dataFactory.create(AccountReport.class);
     List accountSummaries = accountReport.getAccountSummaries();

     CheckingAccount checkingAccount =
          accountDataService.getCheckingAccount(customerID);
     AccountSummary checkingAccountSummary =
          (AccountSummary)dataFactory.create(AccountSummary.class);

checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
);
     checkingAccountSummary.setAccountType("checking");
     checkingAccountSummary.setBalance(fromUSDollarToCurrency
          (checkingAccount.getBalance()));
     accountSummaries.add(checkingAccountSummary);

     SavingsAccount savingsAccount =
          accountDataService.getSavingsAccount(customerID);
     AccountSummary savingsAccountSummary =
          (AccountSummary)dataFactory.create(AccountSummary.class);

savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
     savingsAccountSummary.setAccountType("savings");
     savingsAccountSummary.setBalance(fromUSDollarToCurrency
          (savingsAccount.getBalance()));
     accountSummaries.add(savingsAccountSummary);
```

Formatted: French (France)

```
2644        StockAccount stockAccount =
2645    accountDataService.getStockAccount(customerID);
2646        AccountSummary stockAccountSummary =
2647            (AccountSummary)dataFactory.create(AccountSummary.class);
2648        stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
2649        stockAccountSummary.setAccountType("stock");
2650        float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
2651                    stockAccount.getQuantity();
2652        stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
2653        accountSummaries.add(stockAccountSummary);
2654
2655        return accountReport;
2656        }
2657
2658        @PermitAll
2659        public float fromUSDollarToCurrency(float value) {
2660
2661        if (currency.equals("USD")) return value; else
2662        if (currency.equals("EURO")) return value * 0.8f; else
2663        return 0.0f;
2664        }
2665    }
```

2666 Example 3. Usage of annotated security implementation policy for the java language.

2667 In this example, the implementation class as a whole is marked:

2668 • @RolesAllowed("customers") - indicating that customers have access to the
2669   implementation as a whole

2670 • @RunAs("accountants") – indicating that the code in the implementation runs with the
2671   permissions of accountants

2672 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),
2673 which indicates that this method can be called by both customers and accountants.

2674 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
2675 can be called by any role.

2676

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
        elementFormDefault="qualified">
```

```xml
    <include schemaLocation="sca-core.xsd"/>
```

```xml
    <element name="interface.java" type="sca:JavaInterface"
                substitutionGroup="sca:interface"/>
    <complexType name="JavaInterface">
        <complexContent>
            <extension base="sca:Interface">
                <sequence>
                    <any namespace="##other" processContents="lax"
minOccurs="0"                         maxOccurs="unbounded"/>
                </sequence>
                <attribute name="interface" type="NCName" use="required"/>
                <attribute name="callbackInterface" type="NCName"
use="optional"/>
                <anyAttribute namespace="##any" processContents="lax"/>
            </extension>
        </complexContent>
    </complexType>
</schema>
```

B. Conformance Items

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

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.

2705
2706
2707
2708
2709

Formatted: Font color: Red

Formatted: Font color: Red

Formatted: Font color: Red

B.C.   Acknowledgements

2710

2711 The following individuals have participated in the creation of this specification and are gratefully
2712 acknowledged:

2713 **Participants:**
2714   [Participant Name, Affiliation | Individual Member]
2715   [Participant Name, Affiliation | Individual Member]

2716

~~C.~~D. Non-Normative Text

~~D.~~ **E.** Revision History

[optional; should not be included in OASIS Standards]

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