Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 01, Revision 3

16 December 2008

Deleted: 2

Inserted: 2

Deleted: 03

Deleted: 2

Inserted: 2

Deleted: October

Specification URIs:

This Version:

http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev2.html
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev2.doc
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev2.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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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/.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

Notices

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

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

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

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

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

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

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

Table of Contents

Deleted: 9
Deleted: 9
Deleted: 9
Deleted: 10
Deleted: 11
Deleted: 11
Deleted: 11
Deleted: 11
Deleted: 11
Deleted: 12
Deleted: 12
Deleted: 12
Deleted: 13
Deleted: 13
Deleted: 13
Deleted: 15
Deleted: 15
Deleted: 15
Deleted: 15
Deleted: 15
Deleted: 16
Deleted: 16
Deleted: 16
Deleted: 16
Deleted: 16
Deleted: 17
Deleted: 18
Deleted: 18
Deleted: 18
Deleted: 18
Deleted: 19
Deleted: 19
Deleted: 19
Deleted: 20
Deleted: 21
Deleted: 21
Deleted: 21
Deleted: 21
Deleted: 21
Deleted: 23
Deleted: 24
Deleted: 24
Deleted: 2
Deleted: 2
Inserted: 2
Deleted: WD05
Inserted: 2
Deleted: 03…October ... [1]
Deleted: 25

Deleted: 26
Deleted: 26
Deleted: 27
Deleted: 27
Deleted: 28
Deleted: 29
Deleted: 29
Deleted: 30
Deleted: 30
Deleted: 31
Deleted: 31
Deleted: 31
Deleted: 31
Deleted: 33
Deleted: 33
Deleted: 33
Deleted: 35
Deleted: 35
Deleted: 36
Deleted: 37
Deleted: 37
Deleted: 38
Deleted: 39
Deleted: 39
Deleted: 40
Deleted: 40
Deleted: 41
Deleted: 42
Deleted: 43
Deleted: 45
Deleted: 47
Deleted: 48
Deleted: 49
Deleted: 51
Deleted: 51
Deleted: 53
Deleted: 53
Deleted: 55
Deleted: 56
Deleted: 57
Deleted: 58
Deleted: 59
Deleted: 60
Deleted: 60
Deleted: 60
Deleted: 2
Deleted: 2
Inserted: 2
Deleted: WD05
Inserted: 2
Deleted: 03…October ... [2]

Deleted: 63

Deleted: 67

Deleted: 68

Deleted: 69

Deleted: 70

Deleted: 1 Introduction 7¶
1.1 Terminology 7¶
1.2 Normative References 7¶
1.3 Non-Normative References 8¶
2 Implementation Metadata 9¶
2.1 Service Metadata 9¶
2.1.1 @Service 9¶
2.1.2 Java Semantics of a Remotable Service 9¶
2.1.3 Java Semantics of a Local Service 9¶
2.1.4 @Reference 10¶
2.1.5 @Property 10¶
2.2 Implementation Scopes: @Scope, @Init, @Destroy 10¶
2.2.1 Stateless scope 11¶
2.2.2 Request scope 11¶
2.2.3 Composite scope 11¶
2.2.4 Conversation scope 11¶
3 Interface 12¶
3.1 Java interface element ("interface.java") 12¶
3.2 @Remotable 12¶
3.3 @Conversational 12¶
4 Client API 13¶
4.1 Accessing Services from an SCA Component 13¶
4.1.1 Using the Component Context API 13¶
4.2 Accessing Services from non-SCA component implementations 13¶
4.2.1 ComponentContext 13¶
5 Error Handling 14¶
6 Asynchronous and Conversational Programming 15¶
6.1 @OneWay 15¶ ... [3]

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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/
[POLICY]	SCA Policy Framework, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf
[JSR-250]	Common Annotation for Java Platform specification (JSR-250), http://www.jcp.org/en/jsr/detail?id=250

Deleted: [1].

Deleted: [1].

Deleted: TBD TBD ¶
¶
[1] SCA Assembly Specification¶

Deleted: [2] SDO 2.1 Specification¶

Deleted: [3] JAXB Specification¶

Field Code Changed

Deleted: [4] WSDL Specification¶

Deleted: ¶

Deleted: [5] SCA Policy Framework¶ http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf

Deleted: [6] Common

Deleted: (JSR-250) ¶

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

[JAX-WS] JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224

1.3 Non-Normative References

TBD TBD

Deleted: [7] JAX-WS

Deleted: (JSR-224)¶

Deleted: ¶

Formatted: Bullets and Numbering

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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***)

Deleted: [4]

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.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

2.1.4 @Reference

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

2.1.5 @Property

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

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

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

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

This document defines three scopes:

- STATELESS
- CONVERSATION
- COMPOSITE

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

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

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

@Destroy specifies a method called when the scope ends.

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

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

```

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

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

Deleted: n

Deleted: [1].

Deleted:

Deleted: four

Deleted: ¶

Deleted: REQUEST

Deleted: Note that only no-argument methods may be annotated as lifecycle methods.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
        }
```

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

2.2.1 Stateless scope

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

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

2.2.2 Composite scope

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

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

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

2.2.3 Conversation scope

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

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

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

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

Deleted:

Deleted: <#>Request scope¶
The lifecycle of request scope extends from the point a request on a remotable interface enters the SCA runtime and a thread processes that request until the thread completes synchronously processing the request. During that time, all service requests are delegated to the same implementation instance of a request-scoped component.¶
There are times when a local request scoped service is called without there being a remotable service earlier in the call stack, such as when a local service is called from a non-SCA entity. In these cases, a remote request is always considered to be present, but the lifetime of the request is implementation dependent. For example, a timer event could be treated as a remote request.¶

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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 following snippet shows the schema for the Java interface element.

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

The interface.java element has the following attributes:

- ***interface*** – the fully qualified name of the Java interface

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

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

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.

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

Deleted: ***arguments***

Deleted: ***values***

Deleted: Service Data Objects [2]

Deleted: Service Data Objects [2]

3.2 @Remotable

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

3.3 @Conversational

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

Deleted: the Assembly Specification [1]

Deleted: the Assembly Specification [1]

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

Formatted: Bullets and Numbering

3.4 @Callback

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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!");
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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.

Deleted: n

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.

Deleted: n

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.

Deleted: n

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

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

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

}
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

6.2.2 @EndsConversation

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

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

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

6.3 Passing Conversational Services as Parameters

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

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

```
   interface ComponentContext{
       …
       <B> ServiceReference<B> createSelfReference(Class
                     businessInterface);
       <B> ServiceReference<B> createSelfReference(Class
                     businessInterface, String serviceName);
   }

```

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

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

6.4 Conversational Client

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

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

```

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
public void applyForMortgage(Customer customer, HouseInfo houseInfo,
                             int term)
{
      LoanApplication loanApp;
      loanApp = createApplication(customer, houseInfo);
      loanService.apply(loanApp);
      loanService.lockCurrentRate(term);
}

public boolean isApproved() {
      return loanService.getLoanStatus().equals("approved");
}
public LoanApplication createApplication(Customer customer,
                                         HouseInfo houseInfo) {
      return …;
}
```

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.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

6.6 Conversation ID

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

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

6.6.1 Application Specified Conversation IDs

Deleted: Conversation

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

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

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

6.6.2 Accessing Conversation IDs from Clients

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

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

6.7 Callbacks

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

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

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

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

Deleted: n

6.7.1 Stateful Callbacks

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Conversational;
import org.osoa.sca.annotations.Remotable;
@Remotable
@Conversational
@Callback(MyServiceCallback.class)
public interface MyService {

    void someMethod(String arg);
}

@Remotable
@Conversational
public interface MyServiceCallback {

    void receiveResult(String result);
}
```

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

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

```

@Callback
protected MyServiceCallback callback;

@Reference
protected MyService backendService;

public void someMethod(String arg) {
        backendService.someMethod(arg);
}

public void receiveResult(String result) {
        callback.receiveResult(result);
}
```

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

Deleted: 2
Deleted: 2
Inserted: 2
Deleted: WD05
Inserted: 2
Deleted: 03
Deleted: October

```

private MyService myService;

@Reference
public void setMyService(MyService service) {
         myService = service;
}

public void aClientMethod() {
         ...
      myService.someMethod(arg);
}

public void receiveResult(String result) {
        // code to process the result
}

```

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.

```

private ServiceReference<MyService> myService;

@Reference
public void setMyService(ServiceReference<MyService> service) {
    myService = service;
}

public void aClientMethod() {
    String someKey = "1234";
    ...

    mySevice.setCallbackID(someKey);
    myService.getService().someMethod(arg);
}

@Context RequestContext context;

public void receiveResult(String result) {
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

Deleted: myService

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

```
@Callback
protected MyServiceCallback callback;
```

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

6.7.3 Implementing Multiple Bidirectional Interfaces

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

```
@Callback
protected MyService1Callback callback1;

@Callback
protected MyService2Callback callback2;
```

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

6.7.4 Accessing Callbacks

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

A reference implementing the callback service interface may be obtained using `CallableReference.getService()`.

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

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

public void someMethod() {

   MyCallback myCallback = callback.getCallback();    …

   myCallback.receiveResult(theResult);
}
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

```
public void someMethod() {

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

        …

   myCallback.receiveResult(theResult);
}
```

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

Deleted: (i.e., reference parameters)

```
@Context
protected RequestContext requestContext;

void receiveResult(Object theResult) {

    Object refParams =
            requestContext.getServiceReference().getCallbackID();
        …
}
```

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

Deleted: that was used to send the original request

6.7.5 Customizing the Callback

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

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

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

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

6.7.6 Customizing the Callback Identity

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

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

6.7.7 Bindings for Conversations and Callbacks

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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:

```
package org.osoa.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();


}
```

Deleted: <B> `ServiceReference<B> cast(B target) throws IllegalArgumentException;`

- ***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.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

- ***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.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.

```
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:

```
package org.osoa.sca;

import javax.security.auth.Subject;

public interface RequestContext {

   Subject getSecuritySubject();
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
    String getServiceName();
    <CB> CallableReference<CB> getCallbackReference();
    <CB> CB getCallback();
    <B> CallableReference<B> getServiceReference();

}
```

The RequestContext interface has the following methods:

- ***getSecuritySubject()*** – Returns the JAAS Subject of the current request
- ***getServiceName()*** – Returns the name of the service on the Java implementation the request came in on
- ***getCallbackReference()*** – Returns a callable reference to the callback as specified by the caller
- ***getCallback()*** – Returns a proxy for the callback as specified by the caller
- ***getServiceReference()*** – When invoked during the execution of a service operation, this method MUST return a CallableReference that represents the service that was invoked. When invoked during the execution of a callback operation, this method MUST return a CallableReference that represents the callback that was invoked.

7.3 CallableReference

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

```
package org.osoa.sca;

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

    B getService();
    Class<B> getBusinessInterface();
    boolean isConversational();
    Conversation getConversation();
    Object getCallbackID();
}
```

The CallableReference interface has the following methods:

- **getService()** - Returns a type-safe reference to the target of this reference. The instance returned is guaranteed to implement the business interface for this reference. The value returned is a proxy to the target that implements the business interface associated with this reference.
- **getBusinessInterface()** – Returns the Java class for the business interface associated with this reference.
- **isConversational()** – Returns true if this reference is conversational.
- **getConversation()** – Returns the conversation associated with this reference. Returns null if no conversation is currently active.
- **getCallbackID()** – Returns the callback ID.

7.4 ServiceReference

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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.osoa.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 callaback*)** – Sets the callback object.

7.5 Conversation

The following snippet defines Conversation:

```
package org.osoa.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***.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
package org.osoa.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.osoa.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.osoa.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.osoa.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***.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
package org.osoa.sca;

public class ConversationEndedException extends ServiceRuntimeException {
   …
}
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

8 Java Annotations

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

8.1 @AllowsPassByReference

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

```
package org.osoa.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 {

}
```

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 @Callback

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

```
package org.osoa.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;
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

@Target(TYPE, METHOD, FIELD)
@Retention(RUNTIME)
public @interface Callback {

   Class<?> value() default Void.class;
}
```

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

The @Callback annotation has the following attribute:

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

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

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

Deleted: c

```
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    void someAsyncMethod(String arg);
}
```

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

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Remotable;
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    void someMethod(String arg);
}

@Remotable
public interface MyServiceCallback {

    void receiveResult(String result);
}
```

In this example, the implied component type is:

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >

   <service name="MyService">
         <interface.java interface="somepackage.MyService"
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
                        callbackInterface="somepackage.MyServiceCallback"/>
      </service>
   </componentType>
```

8.3 @ComponentName

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

```
package org.osoa.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.

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

The following snippet shows a component name setter method sample.

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

8.4 @Constructor

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

```
package org.osoa.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 { }
```

Deleted: ¶

Deleted: String[] value() default "";¶

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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.

Deleted: The @Constructor annotation has the following attribute:¶
<#>***value (optional)*** – identifies the property/reference names that correspond to each of the constructor arguments. The position in the array determines which of the arguments are being named. ¶

```
public class HelloServiceImpl implements HelloService {

   public HelloServiceImpl(){
    ...
    }

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

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

8.5 @Context

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

```
package org.osoa.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.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

8.6 @Conversational

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

```
package org.osoa.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 {
}
```

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

The @Conversational annotation has no attributes.

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

```
package services.hello;

import org.osoa.sca.annotations.Conversational;

@Conversational
public interface HelloService {
    void setName(String name);
    String sayHello();
}
```

8.7 @ConversationAttributes

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

```
package org.osoa.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 ConversationAttributes {

   String maxIdleTime() default "";
   String maxAge() default "";
   boolean singlePrincipal() default false;
}
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

- ***maxIdleTime (optional)*** - The maximum time that can pass between successive operations within a single conversation. If more time than this passes, then the container may end the conversation.
- ***maxAge (optional)*** - The maximum time that the entire conversation can remain active. If more time than this passes, then the container may end the conversation.
- ***singlePrincipal (optional)*** – If true, only the principal (the user) that started the conversation has authority to continue the conversation. The default value is false.

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

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

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

```
package service.shoppingcart;

import org.osoa.sca.annotations.ConversationAttributes;

@ConversationAttributes (maxAge="30 days");
public class ShoppingCartServiceImpl implements ShoppingCartService {
   ...
}
```

8.8 @ConversationID

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

```
package org.osoa.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 ConversationID {

}
```

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

The following snippet shows a conversation ID field definition sample.

```
@ConversationID
private String conversationID;
```

The type of the field is not necessarily String.

8.9 @Destroy

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

```
package org.osoa.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;

@Target(METHOD)
@Retention(RUNTIME)
public @interface Destroy {

}
```

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

Deleted: value

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

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

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

8.10 @EagerInit

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

```
package org.osoa.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;
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface EagerInit {

}
```

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

Formatted: Font: Verdana, 9 pt, Complex Script Font: 9 pt

Formatted: Font: Verdana, 9 pt, Bold, Italic, Complex Script Font: 9 pt, Bold, Italic

Formatted: Font: Verdana, 9 pt, Complex Script Font: 9 pt

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

Formatted: Bullets and Numbering

Deleted: ¶

8.11 @EndsConversation

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

```
package org.osoa.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;

@Target(METHOD)
@Retention(RUNTIME)
public @interface EndsConversation {


}
```

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

The @EndsConversation annotation has no attributes.

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

```
package services.shoppingbasket;

import org.osoa.sca.annotations.EndsConversation;

public interface ShoppingBasket {
    void addItem(String itemID, int quantity);

    @EndsConversation
    void buy();
}
```

8.12 @Init

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

```
package org.osoa.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;
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface Init {


}
```

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

Deleted: value

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

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

```
@Init
public void myInitMethod() {
   …
}
```

8.13 @OneWay

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

```
package org.osoa.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;

@Target(METHOD)
@Retention(RUNTIME)
public @interface OneWay {


}
```

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

The @OneWay annotation has no attributes.

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

```
package services.hello;

import org.osoa.sca.annotations.OneWay;

public interface HelloService {
    @OneWay
    void hello(String name);
}
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

8.14 @Property

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

```
package org.osoa.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;
}
```

Deleted: false

Deleted: The @Property annotation is used to denote a Java class field or a setter method 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 setter method input argument.

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.

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 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.

Deleted: the name of the property, defaults to the name of the field of the Java class

Deleted: false

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

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

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

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

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

8.15 @Reference

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

```
package org.osoa.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.

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:

Deleted: denote

Deleted: input argument

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

- ***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.

Deleted: , defaults to the name of the field of the Java class

- ***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 ) {
    ...
}
```

Formatted: Font: Not Bold, Complex Script Font: Bold

Formatted: Complex Script Font: Bold

Deleted: ;

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.

```
package services.hello;

private HelloService helloService;

@Reference(name="helloService", required=true)
public setHelloService(HelloService service) {
        helloService = service;
}

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

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.

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
</componentType>
```

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.

```
   @Reference(name="helloServices", required=true)
   protected List<HelloService> helloServices;

   public void clientMethod() {

         …
         for (int index = 0; index < helloServices.size(); index++) {
               HelloService helloService =
                     (HelloService)helloServices.get(index);
               String result = helloService.hello("Hello World!");
         }
         …
   }
```

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">

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

</componentType>
```

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

8.15.1 Reinjection

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

1. The component MUST NOT be STATELESS scoped.
2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed. Setter injection allows for code in the setter method to perform processing in reaction to a change.
3. If the reference has a conversational interface, then reinjection MUST NOT occur while the conversation is active.

Deleted: or REQUEST

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

If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the reference 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.

A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed. If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the ServiceReference. If the target of a ServiceReference is changed, the reference 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.

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

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

	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.

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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().

Deleted: or REQUEST

8.16 @Remotable

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

```
package org.osoa.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.osoa.sca.annotations.*;

@Remotable
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
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].

Deleted: must

Deleted:

Deleted: w

Deleted: s

Deleted: must

Deleted: 2.0 or 2.1

Deleted: or JAXB 2.0 [3] types

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.sca.annotations.*;

@Remotable
public interface HelloService {

   String hello(String message);
}

package services.hello;

import org.osoa.sca.annotations.*;

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

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

8.17 @Scope

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

```
package org.osoa.sca.annotations;

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

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

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

Formatted: English U.S.

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

Deleted: The default value is 'STATELESS'.

Deleted: REQUEST

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.osoa.sca.annotations.*;

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

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

```

8.18 @Service

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

```
package org.osoa.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;
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
}
```

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.

Deleted: n

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.

Deleted: If a Java implementation needs to realize two services with the same interface, then this is achieved through subclassing of the interface. The subinterface must not add any methods. Both interfaces are listed in the @Service annotation of the Java implementation class.

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

```
package services.hello;

import org.osoa.sca.annotations.Service;

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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.

Deleted: [7]

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.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.

Deleted: supports

Deleted: [2]

Deleted: and the JAXB 2.0 [3] mapping

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">
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
 <part name="ticker" type="xsd:string"/>
</message>

<message name="getPriceResponse">
 <part name="price" type="xsd:float"/>
</message>

<portType name="StockQuote">
 <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
 </operation>
</portType>
```

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

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

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

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

10 Policy Annotations for Java

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

Deleted: [5]

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

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

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

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

Deleted: [6]

10.1 General Intent Annotations

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

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

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

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

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

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

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

Deleted: 2
Deleted: 2
Inserted: 2
Deleted: WD05
Inserted: 2
Deleted: 03
Deleted: October

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

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

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

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

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

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

```
   package org.osoa.sca.annotation;

   import static org.osoa.sca.annotation.Confidentiality.*;

   public class Foo {
      @Requires(CONFIDENTIALITY)
      @Reference
      public void setBar(Bar bar) {
         …
      }
   }
```

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

```
   package org.osoa.sca.annotation;

   import static org.osoa.sca.Constants.*;

   public class Foo {
      @Requires(SCA_PREFIX+”confidentiality”)
      @Reference
      public void setBar(Bar bar) {
         …
      }
   }
```

The formal syntax for the @Requires annotation follows:

```
   @Requires( “qualifiedIntent” | {“qualifiedIntent” [, “qualifiedIntent”]}
```

where

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

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

```
package org.osoa.sca.annotation;
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.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Inherited
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})

public @interface Requires {
   String[] value() default "";
}
```

The SCA_NS constant is defined in the Constants interface:

```
package org.osoa.sca;

public interface Constants {
   String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
   String SCA_PREFIX = "{"+SCA_NS+"}";
}
```

10.2 Specific Intent Annotations

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

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
@Integrity
```

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

```
@Authentication( {“message”, “transport”} )
```

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

The general form of specific intent annotations is:

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

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

```
Intent ::= NCName
qualifiers ::= ”qualifier” | {“qualifier” [, “qualifier”] }
qualifier ::= NCName | NCName/qualifier
```

10.2.1 How to Create Specific Intent Annotations

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

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

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

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

The definition of the @Intent annotation is the following:

```

package org.osoa.sca.annotation;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Retention(RUNTIME)
@Target(ANNOTATION_TYPE)
public @interface Intent {
    String value() default "";
    String targetNamespace() default "";
    String localPart() default "";
}
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

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

```
@Confidentiality({“message”,”transport”})
```

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

The following is the definition of the @Qualifier annotation.

```

package org.osoa.sca.annotation;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Qualifier {
}

```

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

10.3 Application of Intent Annotations

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

- Java class
- Java interface
- Method
- Field

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

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

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

10.3.1 Inheritance And Annotation

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

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

```
package services.hello;
import org.osoa.sca.annotations.Remotable;
import org.osoa.sca.annotations.Integrity;
import org.osoa.sca.annotations.Authentication;

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

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

package services.hello;
import org.osoa.sca.annotations.Remotable;
import org.osoa.sca.annotations.Confidentiality;
import org.osoa.sca.annotations.Authentication;

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

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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

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

The effective intent annotation on the hello method of the HelloService is @Integrity and @Authentication(“message”)

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

```
<?xml version="1.0" encoding="ASCII"?>

<composite  xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
                  name="HelloServiceComposite" >
      <service name=”HelloService” requires=”integrity/transport
            authentication”>
            …
      </service>
      <service name=”HelloChildService” requires=”integrity/transport
            authentication confidentiality/message”>
      …
      </service>
      ...

      <component name="HelloServiceComponent">*
            <implementation.java class="services.hello.HelloService"/>
                        <operation name=”hello” requires=”integrity
                              authentication/message”/>
                        <operation name=”helloThere”
requires=”integrity
                              authentication/transport”/>
      </component>
      <component name="HelloChildServiceComponent">*
            <implementation.java
class="services.hello.HelloChildService" />
            <operation name=”hello”
requires=”confidentiality/transport”/>
            <operation name=”helloThere” requires=” integrity/transport
                  authentication”/>
            <operation name=helloWorld” requires=”authentication”/>
      </component>

      ...

</composite>
```

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

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].

Deleted: [5]

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].

Deleted: [5]

10.6.1 Security Interaction Policy

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

- @Integrity
- @Confidentiality
- @Authentication

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

- message
- transport

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

```
package org.osoa.sca.annotation;

import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD,
                         ElementType.FIELD, ElementType.PARAMETER})
@Intent(Integrity.INTEGRITY)
public @interface Integrity {
    String INTEGRITY = SCA_NS+"integrity";
    String INTEGRITY_MESSAGE = INTEGRITY+".message";
    String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
    @Qualifier
    String[] value() default "";
}


package org.osoa.sca.annotation;

import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
   String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+”.transport”;
   @Qualifier
   String[] value() default "";
}


package org.osoa.sca.annotation;

import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD,
                       ElementType.FIELD, ElementType.PARAMETER})
@Intent(Authentication.AUTHENTICATION)
public @interface Authentication {
    String AUTHENTICATION = SCA_NS+”authentication”;
    String AUTHENTICATION_MESSAGE = AUTHENTICATION+”.message”;
    String AUTHENTICATION_TRANSPORT = AUTHENTICATION+”.transport”;
    @Qualifier
    String[] value() default "";
}
```

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

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

// Interface for ClientService
public interface service.client.ClientService {
      public void clientMethod();
}

// Implementation class for ClientService
package services.client;
```

Deleted: 2
Deleted: WD05
Deleted: 2
Inserted: 2
Inserted: 2
Deleted: 03
Deleted: October
Formatted: English U.S.

```
import services.hello.HelloService;

import org.osoa.sca.annotations.*;

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


    private HelloService helloService;

    @Reference(name="helloService", required=true)
    @Integrity(“message”)
    @Authentication(“message”)
    public void setHelloService(HelloService service) {
        helloService = service;
    }

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

Example 1. Usage of annotated intents on a reference.

10.6.2 Security Implementation Policy

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

- DenyAll

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

- DeclareRoles

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

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

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

Deleted: [5]

10.6.2.1 Annotated Implementation Policy Example

The following is an example showing annotated security implementation policy:

```
package services.account;
@Remotable
public interface AccountService {
      AccountReport getAccountReport(String customerID);
}
```

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

```
package services.account;
import java.util.List;
import commonj.sdo.DataFactory;
import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.RolesAllowed;
import org.osoa.sca.annotations.RunAs;
import org.osoa.sca.annotations.PermitAll;
import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;
@RolesAllowed(“customers”)
@RunAs(“accountants” )
public class AccountServiceImpl implements AccountService {

   @Property
   protected String currency = "USD";

   @Reference
   protected AccountDataService accountDataService;
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
   @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);

    StockAccount stockAccount =
accountDataService.getStockAccount(customerID);
    AccountSummary stockAccountSummary =
         (AccountSummary)dataFactory.create(AccountSummary.class);
    stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
    stockAccountSummary.setAccountType("stock");
    float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
                     stockAccount.getQuantity();
    stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
    accountSummaries.add(stockAccountSummary);
```

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

```
    return accountReport;
   }

   @PermitAll
   public float fromUSDollarToCurrency(float value) {

    if (currency.equals("USD")) return value; else
    if (currency.equals("EURO")) return value * 0.8f; else
    return 0.0f;
   }
}
```

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

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

- @RolesAllowed(“customers”) - indicating that customers have access to the implementation as a whole
- @RunAs(“accountants”) – indicating that the code in the implementation runs with the permissions of accountants

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

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

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

Formatted: Bullets and Numbering

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

```
<?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">

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

    <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>
```

Deleted: http://www.osoa.org/xmlns/sca/1.0

Deleted: http://www.osoa.org/xmlns/sca/1.0

Formatted: German Germany

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

Formatted: Bullets and Numbering

B. Acknowledgements

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

Participants:

[Participant Name, Affiliation | Individual Member]
[Participant Name, Affiliation | Individual Member]

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

C. Non-Normative Text

Formatted: Bullets and Numbering

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

D. 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

Deleted: 75(?),

Deleted: 2

Deleted: 2

Inserted: 2

Deleted: WD05

Inserted: 2

Deleted: 03

Deleted: October

Page 1: [1] Deleted **user** **12/11/2008 9:35 PM**

03

Page 1: [1] Deleted **user** **12/11/2008 9:35 PM**

October

Page 1: [2] Deleted **user** **12/11/2008 9:35 PM**

03

Page 1: [2] Deleted **user** **12/11/2008 9:35 PM**

October

Page 6: [3] Deleted **user** **12/12/2008 4:57 PM**