Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 03+Issue1 rev 5

22 June 2009

Deleted: 2

Deleted: 19

Deleted: 6

Deleted: March

Specification URIs:
This Version:
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.html
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.doc
http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.pdf (normative)

Previous Version:

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

Latest Approved Version:

Technical Committee:
OASIS Service Component Architecture / J (SCA-J) TC

Chair(s):

Simon Nash,	IBM
MIchael Rowley,	BEA Systems
Mark Combellack,	Avaya

Editor(s):

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

Related work:
This specification replaces or supersedes:

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

This specification is related to:

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

Declared XML Namespace(s):

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

Abstract:

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

Specifically, this specification covers:

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

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

Status:

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

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

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

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

Notices

Table of Contents

Deleted: 36
Deleted: 37
Deleted: 37
Deleted: 37
Deleted: 38
Deleted: 39
Deleted: 40
Deleted: 41
Deleted: 41
Deleted: 42
Deleted: 42
Deleted: 43
Deleted: 43
Deleted: 44
Deleted: 45
Deleted: 45
Deleted: 46
Deleted: 47
Deleted: 48
Deleted: 50
Deleted: 52
Deleted: 53
Deleted: 54
Deleted: 55
Deleted: 57
Deleted: 57
Deleted: 59
Deleted: 60
Deleted: 60
Deleted: 60
Deleted: 61
Deleted: 62
Deleted: 67
Deleted: 69
Deleted: 75

Deleted: 76

Deleted: 77

1 Introduction

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

Specifically, this specification covers:

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

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

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

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

Comment [ME1]: This sentence needs to be removed

1.1 Terminology

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

1.2 Normative References

[RFC2119]	S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.
[ASSEMBLY]	SCA Assembly Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf
[SDO]	SDO 2.1 Specification, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf
[JAX-B]	JAXB 2.1 Specification, http://www.jcp.org/en/jsr/detail?id=222
[WSDL]	WSDL Specification, WSDL 1.1: http://www.w3.org/TR/wsdl, WSDL 2.0: http://www.w3.org/TR/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
[JAX-WS]	JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224
[JAVABEANS]	JavaBeans 1.01 Specification, http://java.sun.com/javase/technologies/desktop/javabeans/api/

1.3 Non-Normative References

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

2 Implementation Metadata

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

2.1 Service Metadata

2.1.1 @Service

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

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

2.1.2 Java Semantics of a Remotable Service

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

Deleted: Remotable Services MUST NOT make use of ***method overloading***.

Deleted: Remotable Services MUST NOT make use of ***method overloading***.

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

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

2.1.3 Java Semantics of a Local Service

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

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

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

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

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

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

2.1.4 @Reference

Accessing a service using reference injection is done by defining a field, a setter method 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 two scopes:

- STATELESS
- COMPOSITE

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

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

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

@Destroy specifies a method called when the scope ends.

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

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

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

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

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

2.2.1 Stateless scope

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

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. [JCA20002] In addition, within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method. [JCA20003] 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

For a composite scope implementation instance, the SCA runtime MUST ensure that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite. [JCA20004] 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.

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

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 the SCA runtime MUST NOT perform any synchronization. [JCA20007]

Deleted: When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.

Deleted: When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.

Deleted: If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.

Deleted: If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.

3 Interface

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

3.1 Java interface element – <interface.java>

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

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

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

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

The interface.java element has the following attributes:

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

Deleted: The value of the @interface attribute MUST be the fully qualified name of the Java interface class

Deleted: The value of the @interface attribute MUST be the fully qualified name of the Java interface class

Deleted: The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks

Deleted: The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks

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

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

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

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

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

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 @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.

4 Client API

This section describes how SCA services can 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 can 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 SCAClient Interface and Related Classes

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

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

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

```
import org.oasisopen.sca.client.SCAClient;
import org.oasisopen.sca.client.SCAClientFactory;
import com.foo.HelloService;

public void someMethod() {

        String serviceURI = "SomeHelloServiceURI";
        URI domainURI = new URI("SomeDomainURI");

        ...
        SCAClient scaClient = SCAClientFactory.newInstance();
        HelloService helloService =
            scaClient.getService(HelloService.class,
                                 serviceURI, domainURI);
        String reply = helloService.sayHello("Mark");
        ...
    }
```

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

Deleted: ComponentContext

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

Deleted: d

Deleted: ¶

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

Formatted: Bullets and Numbering

5 Error Handling

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

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

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

6 Asynchronous Programming

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

- support for non-blocking method calls
- callbacks

Each of these topics is discussed in the following sections.

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

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

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

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

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

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

6.2.1 Using Callbacks

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

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

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

```
package somepackage;
import org.osoa.sca.annotation.Callback;
import org.osoa.sca.annotation.Remotable;
@Remotable
@Callback(QuotationCallback.class)
public interface Quotation {h
    double requestQuotation(String productCode, int quantity);
}

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

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

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

```
@Callback
protected QuotationCallback callback;

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

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

```
public class ClientImpl implements ClientService, QuotationCallback {

    private QuotationService myService;

    @Reference
    public void setMyService(QuotationService service) {
        myService = service;
```

```
        }

        public void aClientMethod() {
            ...
            double quote = myService.requestQuotation(“AB123”, 2000);
            ...
        }

        public String getState() {
            return “TX”;
        }
        public String getZipCode() {
            return “78746”;
        }
        public String getCreditRating() {
            return “AA”;
        }
    }
```

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

6.2.2 Callback Instance Management

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

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

6.2.3 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.

6.2.4 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 of type ***ServiceReference*** with the ***@Callback*** annotation.

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

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

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

public void someMethod() {

   MyCallback myCallback = callback.getCallback();    …

   myCallback.receiveResult(theResult);
}
```

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

Alternatively, a callback can 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:

```
@Context
protected RequestContext requestContext;

void receiveResult(Object theResult) {

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

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

7 Policy Annotations for Java

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

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

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

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

This specification supports using the Common 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 supports consistent annotation and Java class inheritance relationships.

7.1 General Intent Annotations

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

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

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

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

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

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

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

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 com.foo;

import static org.oasisopen.sca.annotation.Confidentiality.*;
import static org.oasisopen.sca.annotation.Reference;
import static org.oasisopen.sca.annotation.Requires;

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

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

```
package com.foo;

import static org.oasisopen.sca.Constants.*;
import static org.oasisopen.sca.annotation.Reference;
import static org.oasisopen.sca.annotation.Requires;

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

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

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

where

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

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

7.2 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) annotation can also be specified with the @Confidentiality specific intent annotation. The specific intent annotation for the "integrity" security intent is:

```
@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   ::= NCName ('.' qualifier)?
```

7.2.1 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 a specific intent annotation. [JCA70001]

Deleted: SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.

Deleted: SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific 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, for 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 is also possible to use one or more of them as parameters to the specific intent annotation.

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

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

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

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 needs to be marked with the @Qualifier annotation. The @Qualifier tells SCA that the value of the attribute is 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 exist. 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 annotation is attached.

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

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.

7.3 Application of Intent Annotations

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

- Java class
- Java interface
- Method
- Field
- Constructor parameter

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 base intent name.

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.

7.3.1 Inheritance And Annotation

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

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

```
package services.hello;
import org.oasisopen.sca.annotation.Remotable;
import org.oasisopen.sca.annotation.Integrity;
import org.oasisopen.sca.annotation.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.oasisopen.sca.annotation.Remotable;
import org.oasisopen.sca.annotation.Confidentiality;
import org.oasisopen.sca.annotation.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 of the ***HelloChildService*** is Integrity(“transport”), @Authentication, and @Confidentiality(“message”).

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.

7.4 Relationship of Declarative And Annotated Intents

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

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

7.5 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>”)* )
```

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

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

- Java class
- Java interface
- Method
- Field
- Constructor parameter

7.6 Security Policy Annotations

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

7.6.1 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:

- @Integrity
- @Confidentiality
- @Authentication

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

- message
- transport

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

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.

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

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

// Implementation class for ClientService
package services.client;

import services.hello.HelloService;
import org.oasisopen.sca.annotation.*;

@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.

7.6.2 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 executes 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.

- 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].

7.6.2.1 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);
      float fromUSDollarToCurrency(float value);
}
```

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.oasisopen.sca.annotation.Property;
import org.oasisopen.sca.annotation.Reference;
import org.oasisopen.sca.annotation.RolesAllowed;
import org.oasisopen.sca.annotation.RunAs;
import org.oasisopen.sca.annotation.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;
   @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);

        return accountReport;
       }

       @PermitAll
       public float fromUSDollarToCurrency(float value) {

        if (currency.equals("USD")) return value;
        if (currency.equals("EURO")) return value * 0.8f;
        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.

8 Java API

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

8.1 Component Context

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

```
package org.oasisopen.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 ServiceReference<B>> R cast(B target)
                throws  IllegalArgumentException;

   RequestContext getRequestContext();


}
```

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

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

A component can access its component context by defining a field or setter method typed by ***org.oasisopen.sca.ComponentContext*** and annotated with ***@Context***. To access 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");
}
```

8.2 Request Context

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

```
package org.oasisopen.sca;

import javax.security.auth.Subject;

public interface RequestContext {

   Subject getSecuritySubject();

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

Deleted: ¶
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.¶

Formatted: Bullets and Numbering

```
}
```

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 service reference to the callback as specified by the caller. This method returns null when called for a service request whose interface is not bidirectional or when called for a callback request.
- ***getCallback()*** – Returns a proxy for the callback as specified by the caller. Similar to the getCallbackReference() method, this method returns null when called for a service request whose interface is not bidirectional or when called for a callback request.
- ***getServiceReference()*** – When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked. [JCA80003]

Comment [ME2]: Need a reference to JAAS here

Comment [ME3]: What happens if there is no JAAS subject?

Deleted: When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.

Deleted: When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.

8.3 ServiceReference

ServiceReferences can 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.oasisopen.sca;

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

    B getService();
    Class<B> getBusinessInterface();
}
```

The ServiceReference 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.

8.4 ServiceRuntimeException

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

```
package org.oasisopen.sca;

public class ServiceRuntimeException extends RuntimeException {
    …
}
```

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

8.5 ServiceUnavailableException

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

```
package org.oasisopen.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    …
}
```

This exception signals problems in the interaction with remote services. These are exceptions that can 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

8.6 InvalidServiceException

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

```
package org.oasisopen.sca;

public class InvalidServiceException extends ServiceRuntimeException {
    …
}
```

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

8.7 Constants

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

```
package org.oasisopen.sca;

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

8.8 SCAClient Interface

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

The following shows the ***SCAClient*** interface:

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

public interface SCAClient {

    <T> T getService(Class<T> interfaze,
                     String serviceURI,
                     URI domainURI)
```

```
        throws NoSuchServiceException, NoSuchDomainException;
    }
```

getService method:

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

Returns:

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

Parameters:

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

Exceptions:

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

8.9 SCAClientFactory Class

The SCAClientFactory class provides the means for client code to obtain an object which implements the SCAClient interface which is used in turn to obtain a proxy reference object to a service within an SCA Domain.

The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods which the client can invoke in order to obtain a object implementing the SCAClient interface.

The SCAClientFactory class is as follows:

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

import java.util.Properties;
import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

public abstract class SCAClientFactory {

    protected static SCAClientFactoryFinderImpl defaultFactoryFinder;

    public static SCAClient newInstance() {
        return newInstance(null, null);
    }


    public static SCAClient newInstance(Properties properties) {
        return newInstance(properties, null);
    }

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

Formatted: Indent: Before: 0.25"

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

    public static SCAClient newInstance(Properties properties,
                                        ClassLoader classLoader) {
        final SCAClientFactoryFinderImpl finder =
            defaultFactoryFinder != null ? defaultFactoryFinder :
            new SCAClientFactoryFinderImpl();
        final SCAClientFactory factory
            = finder.find(properties, classLoader);
        return factory.createSCAClient();
    }

    protected abstract SCAClient createSCAClient();
}
```

Deleted: ¶

newInstance() method:

Obtains a object implementing the SCAClient interface.

Returns:

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

Parameters:

- ***none***

Exceptions:

- ***none***

newInstance(Properties) method:

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

Returns:

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

Parameters:

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

Exceptions:

- ***none***

newInstance(Classloader) method:

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

Returns:

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

Parameters:

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

Exceptions:

- ***none***

newInstance(Properties, Classloader) method:

Obtains a object implementing the SCAClient interface using a specified set of properties and a specified classloader.

Returns:

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

Parameters:

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

Exceptions:

- ***none***

defaultFactory protected field:

Provides a means by which a provider of an SCAClientFactory implementation can inject its implementation subclass into the abstract SCAClientFactory class - once this is done, future invocations of the SCAClientFactory return the implementation subclass.

8.10 SCAClientFactoryFinder Interface

The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can create alternative implementations of this interface that use different class loading or lookup mechanisms.

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

import java.util.Properties;
public interface SCAClientFactoryFinder {

     SCAClientFactory find(Properties properties,
                           ClassLoader classLoader);
}
```

8.11 SCAClientFactoryFinderImpl Class

This class is a default implementation of an SCAClientFactoryFinder, which is used to find an implementation of the SCAClientFactory interface, as used to obtain an SCAClient object for use by a client. SCA runtime providers can replace this implementation with their own version.

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

import org.oasisopen.sca.client.SCAClientFactoryFinder;

import java.io.BufferedReader;
import java.io.Closeable;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.Properties;

import org.oasisopen.sca.SCARuntimeException;
import org.oasisopen.sca.client.SCAClientFactory;
```

Formatted: Bullets and Numbering

Formatted: Body Text,Body Text Char,Body Text Char1 Char1,Body Text Char Char Char1,Body Text Char1 Char1 Char Char,Body Text Char Char Char1 Char Char,Body Text Char1 Char1 Char Char Char Char,Body Text Char Char Char1 Char Char Char Char,Body Text Char1

Formatted: Indent: Before: 0.25"

Formatted: Body Text,Body Text Char,Body Text Char1 Char1,Body Text Char Char Char1,Body Text Char1 Char1 Char Char,Body Text Char Char Char1 Char Char,Body Text Char1 Char1 Char Char Char Char,Body Text Char Char Char1 Char Char Char Char,Body Text Char1

Formatted: Bullets and Numbering

Formatted: Body Text,Body Text Char,Body Text Char1 Char1,Body Text Char Char Char1,Body Text Char1 Char1 Char Char,Body Text Char Char Char1 Char Char,Body Text Char1 Char1 Char Char Char Char,Body Text Char Char Char1 Char Char Char Char,Body Text Char1

Formatted: Indent: Before: 0.25"

Formatted: French France

```
1268
1269    public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder
1270    {
1271
1272        private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
1273                SCAClientFactory.class.getName();
1274
1275        private static final String
1276                SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
1277                = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
1278
1279        public SCAClientFactoryFinderImpl() {
1280        }
1281
1282        public SCAClientFactory find(Properties properties,
1283                                     ClassLoader classLoader) {
1284            if (classLoader == null) {
1285                classLoader = getThreadContextClassLoader();
1286            }
1287            final String factoryImplClassName =
1288                discoverProviderFactoryImplClass(properties, classLoader);
1289            final Class<? extends SCAClientFactory> factoryImplClass
1290                = loadProviderFactoryClass(factoryImplClassName,
1291                                           classLoader);
1292            final SCAClientFactory factory =
1293                instantiateSCAClientFactoryClass(factoryImplClass);
1294            return factory;
1295       }
1296
1297        private static ClassLoader getThreadContextClassLoader() {
1298            final ClassLoader threadClassLoader =
1299                Thread.currentThread().getContextClassLoader();
1300            return threadClassLoader;
1301        }
1302
1303        private static String
1304            discoverProviderFactoryImplClass(Properties properties,
1305                                             ClassLoader classLoader)
1306            throws SCARuntimeException {
1307            String providerClassName =
1308             checkPropertiesForSPIClassName(properties);
1309            if (providerClassName != null) {
1310                return providerClassName;
1311            }
1312
1313            providerClassName =
1314             checkPropertiesForSPIClassName(System.getProperties());
1315            if (providerClassName != null) {
1316                return providerClassName;
1317            }
1318
1319            providerClassName =
1320                checkMETAINFServicesForSIPClassName(classLoader);
1321            if (providerClassName == null) {
1322                throw new SCARuntimeException(
1323                    "Failed to find implementation for SCAClientFactory");
1324            }
1325
```

Formatted: Indent: Before: 0.58"

Formatted: Indent: Before: 0.58", First line: 0.42"

Formatted: Indent: Before: 0.25"

```
1326            return providerClassName;
1327        }
1328
1329        private static String
1330            checkPropertiesForSPIClassName(Properties properties) {
1331            if (properties == null) {
1332                return null;
1333            }
1334
1335            final String providerClassName =
1336             properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
1337            if (providerClassName != null && providerClassName.length() > 0) {
1338                return providerClassName;
1339            }
1340
1341            return null;
1342        }
1343
1344        private static String checkMETAINFServicesForSIPClassName(ClassLoader
1345                                                                  cl)
1346        {
1347            final URL url =
1348             cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
1349            if (url == null) {
1350                return null;
1351            }
1352
1353            InputStream in = null;
1354            try {
1355                in = url.openStream();
1356                BufferedReader reader = null;
1357                try {
1358                    reader =
1359                        new BufferedReader(new InputStreamReader(in,
1360                                                                 "UTF-8"));
1361
1362                    String line;
1363                    while ((line = readNextLine(reader)) != null) {
1364                        if (!line.startsWith("#") && line.length() > 0) {
1365                            return line;
1366                        }
1367                    }
1368
1369                    return null;
1370                } finally {
1371                    closeStream(reader);
1372                }
1373            } catch (IOException ex) {
1374                throw new SCARuntimeException(
1375                        "Failed to discover SCAClientFactory provider", ex);
1376            } finally {
1377                closeStream(in);
1378            }
1379        }
1380
1381        private static String readNextLine(BufferedReader reader)
1382            throws IOException {
1383
```

```
1384            String line = reader.readLine();
1385            if (line != null) {
1386                line = line.trim();
1387            }
1388            return line;
1389        }
1390
1391        private static Class<? extends SCAClientFactory>
1392            loadProviderFactoryClass(String factoryImplClassName,
1393                                     ClassLoader classLoader)
1394            throws SCARuntimeException {
1395
1396            try {
1397                final Class<?> providerClass =
1398                   classLoader.loadClass(factoryImplClassName);
1399                final Class<? extends SCAClientFactory> providerFactoryClass =
1400                    providerClass.asSubclass(SCAClientFactory.class);
1401                return providerFactoryClass;
1402            } catch (ClassNotFoundException ex) {
1403                throw new SCARuntimeException(
1404                   "Failed to load SCAClientFactory implementation class "
1405                   + factoryImplClassName, ex);
1406            } catch (ClassCastException ex) {
1407                throw new SCARuntimeException(
1408                        "Loaded SCAClientFactory implementation class "
1409                        + factoryImplClassName
1410                       + " is not a subclass of "
1411                       + SCAClientFactory.class.getName() , ex);
1412            }
1413        }
1414
1415        private static SCAClientFactory
1416            instantiateSCAClientFactoryClass(
1417                    Class<? extends SCAClientFactory> factoryImplClass)
1418            throws SCARuntimeException {
1419
1420            try {
1421                final SCAClientFactory provider =
1422                                        factoryImplClass.newInstance();
1423                return provider;
1424            } catch (Throwable ex) {
1425                throw new SCARuntimeException(
1426                   "Failed to instantiate SCAClientFactory implementation
1427                    class " + factoryImplClass, ex);
1428            }
1429        }
1430
1431        private static void closeStream(Closeable closeable) {
1432            if (closeable != null) {
1433                try{
1434                    closeable.close();
1435                } catch (IOException ex) {
1436                    throw new SCARuntimeException("Failed to close stream",
1437                                                  ex);
1438                }
1439            }
1440        }
1441    }
```

Formatted: Body Text,Body Text Char,Body Text Char1 Char1,Body Text Char Char Char1,Body Text Char1 Char1 Char Char,Body Text Char Char Char1 Char Char,Body Text Char1 Char1 Char Char Char Char,Body Text Char Char Char1 Char Char Char Char,Body Text Char1

8.12 NoSuchDomainException

The following shows the NoSuchDomainException:

```
package org.oasisopen.sca;

public class NoSuchDomainException extends Exception {
    ...
}
```

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

8.13 NoSuchServiceException

The following shows the NoSuchServiceException:

```
package org.oasisopen.sca;

public class NoSuchServiceException extends Exception {
    ...
}
```

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

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

9 Java Annotations

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

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

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

9.1 @AllowsPassByReference

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

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

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

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

}
```

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) {
    …
}
```

9.2 @Authentication

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

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

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

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

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

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

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

9.3 @Callback

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

```
package org.oasisopen.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.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
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 by specifying the Java class object of the callback interface as an attribute.

The @Callback annotation has the following attribute:

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

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

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

```
package somepackage;
import org.oasisopen.sca.annotation.Callback;
import org.oasisopen.sca.annotation.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"
                  callbackInterface="somepackage.MyServiceCallback"/>
   </service>
</componentType>
```

9.4 @ComponentName

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

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

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) {
 //…
}
```

9.5 @Confidentiality

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

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

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

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

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

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

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

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

9.6 @Constructor

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

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

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

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

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

Comment [ME4]: There also needs to be a normative statement that at most 1 constructor can be annotated with @Constructor

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

```
public class HelloServiceImpl implements HelloService {

   public HelloServiceImpl(){
    ...
    }

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

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

9.7 @Context

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

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

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

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

```
}
```

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

The @Context annotation has no attributes.

The following snippet shows a ComponentContext field definition sample.

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

The following snippet shows a RequestContext field definition sample.

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

9.8 @Destroy

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

```
package org.oasisopen.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;

@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. A method annotated with @Destroy MAY have any access modifier and MUST have a void return type and no arguments. [JCA90004]

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

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

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

9.9 @EagerInit

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

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

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 EagerInit {

}
```

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

9.10 @Init

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

```
package org.oasisopen.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;

@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. A method marked with the @Initi annotation MAY have any access modifier and MUST have a void return type and no arguments. [JCA90008]

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

Deleted: If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.

Deleted: If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.

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

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

9.11 @Integrity

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

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

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

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

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

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

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

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

9.12 @Intent

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

```
package org.oasisopen.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;

@Target({ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface Intent {
    /**
     * The qualified name of the intent, in the form defined by
     * {@link javax.xml.namespace.QName#toString}.
     * @return the qualified name of the intent
     */
    String value() default "";

    /**
```

```
         * The XML namespace for the intent.
         * @return the XML namespace for the intent
         */
        String targetNamespace() default "";

        /**
         * The name of the intent within its namespace.
         * @return name of the intent within its namespace
         */
        String localPart() default "";
    }
```

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

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

9.13 @OneWay

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

```
package org.oasisopen.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;

@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.

Comment [ME5]: Needs recasting in a normative form of statement

The @OneWay annotation has no attributes.

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

```
package services.hello;

import org.oasisopen.sca.annotation.OneWay;

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

9.14 @PolicySets

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

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

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

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

@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
public @interface PolicySets {
    /**
     * Returns the policy sets to be applied.
     *
     * @return the policy sets to be applied
     */
    String[] value() default "";
}
```

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

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

9.15 @Property

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

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

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

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

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

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

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

Properties can 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. [JCA90012] 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.

Deleted: the @Property annotation MUST NOT be used on a class field that is declared as final.

Deleted: the @Property annotation MUST NOT be used on a class field that is declared as final.

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

The @Property annotation has the following attributes:

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

The following snippet shows a property field definition sample.

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

The following snippet shows a property setter sample

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

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

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;
}
...
```

9.16 @Qualifier

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

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

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

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

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

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

@Target(METHOD)
@Retention(RUNTIME)
public @interface Qualifier {
}
```

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

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

9.17 @Reference

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

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

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

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

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

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

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

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

The @Reference annotation has the following attributes:

- ***name : String (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 @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present. [JCA90018]

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

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 ) {
    ...
}
```

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>

</componentType>
```

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

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

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 needs to 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>
```

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

9.17.1 Reinjection

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

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

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

Deleted: If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the compo ... [1]

Deleted: If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the compo ... [2]

Deleted: If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the compo ... [3]

Deleted: An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null

Deleted: An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null

Deleted: An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty ... [4]

Deleted: An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty ... [5]

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.

[JCA90025]

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

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. [JCA90026]

If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made. [JCA90029] 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. [JCA90030] If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference. [JCA90031] If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference. [JCA90032] If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made. [JCA90033] 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. [JCA90034] If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException. [JCA90035] If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service. [JCA90036]

The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This means that in the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring. [JCA90037] In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents. [JCA90038] A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component. [JCA90039]

	Effect on		
Change event	Injected Reference or ServiceReference	Existing ServiceReference Object**	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	can be reinjected (if other conditions* apply). If not reinjected, then it	continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.

Deleted: 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 ... [6]

Deleted: In order for reinjection to occur, the following MUST be true:¶ 1. . The component MUS ... [7]

Deleted: If the target service of the reference is changed, the reference MAY continue to work ... [8]

Deleted: If the target service of the reference is changed, the reference MAY continue to work ... [9]

Deleted: If the target service of a ServiceReference is changed, the referer ... [10]

Deleted: If the target service of a ServiceReference is changed, the referer ... [11]

Deleted: A reference or ServiceReference accessed through the component context by calling ... [12]

Deleted: A reference or ServiceReference accessed through the component context by calling ... [13]

Deleted: If the target of a reference or ServiceReference accessed through the compon ... [14]

Deleted: If the target of a reference or ServiceReference accessed through the compon ... [15]

Deleted: If the target service of a reference or ServiceReference accessed through the compon ... [16]

Deleted: If the target service of a reference or ServiceReference accessed through the compon ... [17]

Deleted: In cases where the contents of a reference array or collection change when the wiring cha ... [18]

Deleted: In cases where the contents of a reference array or collection change when the wiring cha ... [19]

	continues to work as if the reference target was not changed.		
Target service undeployed	Business methods throw InvalidServiceException.	Business methods throw InvalidServiceException.	Result is a reference to the undeployed service. Business methods throw InvalidServiceException.
Target service becomes unavailable	Business methods throw ServiceUnavailableException	Business methods throw ServiceUnavailableException	Result is be a reference to the unavailable service. Business methods throw ServiceUnavailableException.
Target service changed	might continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	might continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	Result is a reference to the changed service.

* Other conditions:

The component cannot be STATELESS scoped.

The reference has to use either field-based injection or setter injection. References that are injected through constructor injection cannot be changed.

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

9.18 @Remotable

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

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

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. [JCA90040]

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.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {

   String hello(String message);
}
```

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

Complex data types exchanged via remotable service interfaces need to 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 can be JAXB [JAX-B] types or they can be Service Data Objects (SDOs) [SDO].

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

Implementations of remotable services can modify input data during or after an invocation and can 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.oasisopen.sca.annotation.*;

@Remotable
public interface HelloService {

   String hello(String message);
}

package services.hello;

import org.oasisopen.sca.annotation.*;

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

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

9.19 @Requires

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

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

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

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

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

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

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

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

9.20 @Scope

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

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

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

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

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

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

The @Scope annotation has the following attribute:

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

The default value is STATELESS.

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

```
package services.hello;
```

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

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

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

9.21 @Service

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

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

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;
}
```

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

Deleted: An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.

Deleted: An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.

The @Service annotation has the following attributes:

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

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

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

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

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

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

```
package services.hello;

import org.oasisopen.sca.annotation.Service;

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

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

10 WSDL to Java and Java to WSDL

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

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

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

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.

10.1 JAX-WS Client Asynchronous API for a Synchronous Service

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

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

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

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

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

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

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

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

Deleted: For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.

Deleted: For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.

Deleted: The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.

Deleted: The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.

Deleted: For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.

Deleted: For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.

c. a return type of Future<?>

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

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

```
<!-- WSDL extract -->
<message name="getPrice">
 <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. [JCA100009] 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.

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>
```

B. Java Classes and Interfaces

B.1 SCAClient Classes and Interfaces

B.1.1 SCAClient Interface

```
/*
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
 * OASIS trademark, IPR and other policies apply.
 */
package org.oasisopen.sca.client;

import java.net.URI;

import org.oasisopen.sca.NoSuchDomainException;
import org.oasisopen.sca.NoSuchServiceException;

/**
 * Client side interface that can be used to lookup SCA Services within
 * a SCA domain.
 * <p>
 * The SCAClientFactory is used to obtain an implementation instance of
 * the SCAClient.
 *
 * @see SCAClientFactory
 * @author OASIS Open
 */
public interface SCAClient {

    /**
     * Returns a reference proxy that implements the business interface <T>
     * of a service in a domain
     *
     * @param serviceURI the relative URI of the target service. Takes the
     * form componentName/serviceName.
     * Can also take the extended form componentName/serviceName/bindingName
     * to use a specific binding of the target service
     *
     * @param domainURI the URI of an SCA Domain.
     * @param interfaze The business interface class of the service in the
     * domain
     * @param <T> The business interface class of the service in the domain
     *
     * @return a proxy to the target service, in the specified SCA Domain
     * that implements the business interface <B>.
     * @throws NoSuchServiceException Service requested was not found
     * @throws NoSuchDomainException Domain requested was not found
     */
    <T> T getService(Class<T> interfaze, String serviceURI, URI domainURI)
        throws NoSuchServiceException, NoSuchDomainException;
```

Formatted: French France

Formatted: French France

```
}
```

Deleted: D

B.1.2 SCAClientFactory Class

SCA provides an abstract reference implementation of the SCAClientFactory class. Vendors can provide a subclass of this class which provides the capability of creating objects that implement the SCAClient interface suitable for linking to services in their SCA runtime.

```
/*
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
 * OASIS trademark, IPR and other policies apply.
 */
package org.oasisopen.sca.client;

import java.util.Properties;

import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

/**
 * The SCAClientFactory can be used by non-SCA managed code to
 * lookup services that exist in a SCADomain.
 *
 * @see SCAClientFactoryFinderImpl
 * @see SCAClient
 *
 * @author OASIS Open
 */

public abstract class SCAClientFactory {

    /**
     * The SCAClientFactoryFinder.
     * A Vendor may use reflection to inject a their own version of an
     * SCAClientFactoryFinder instance that will be used in the
     * newInstance() methods rather than using the default one provided by
     * SCAClientFactoryFinderImpl
     */
    protected static SCAClientFactoryFinderImpl defaultFactoryFinder;

    /**
     * Creates a new instance of the SCAClient that can be
     * used to lookup SCA Services.
     *
     * @return A new SCAClient
     */
    public static SCAClient newInstance() {
        return newInstance(null, null);
    }

    /**
     * Creates a new instance of the SCAClient that can be
     * used to lookup SCA Services.
     *
     * @param properties   Properties that may be used when
     * creating a new instance of the SCAClient
     * @return A new SCAClient instance
```

```
     */
    public static SCAClient newInstance(Properties properties) {
        return newInstance(properties, null);
    }

    /**
     * Creates a new instance of the SCAClient that can be
     * used to lookup SCA Services.
     *
     * @param classLoader   ClassLoader that may be used when
     * creating a new instance of the SCAClient
     * @return A new SCAClient instance
     */
    public static SCAClient newInstance(ClassLoader classLoader) {
        return newInstance(null, classLoader);
    }

    /**
     * Creates a new instance of the SCAClient that can be
     * used to lookup SCA Services.
     *
     * @param properties    Properties that may be used when
     * creating a new instance of the SCAClient
     * @param classLoader   ClassLoader that may be used when
     * creating a new instance of the SCAClient
     * @return A new SCAClient instance
     */
    public static SCAClient newInstance(Properties properties,
                                        ClassLoader classLoader) {
        final SCAClientFactoryFinderImpl finder =
            defaultFactoryFinder != null ? defaultFactoryFinder :
            new SCAClientFactoryFinderImpl();
        final SCAClientFactory factory
            = finder.find(properties, classLoader);
        return factory.createSCAClient();
    }

    /**
     * This method is invoked to create a new SCAClient instance.
     *
     * @return A new SCAClient instance
     */
    protected abstract SCAClient createSCAClient();
},
```

Deleted: protected

Formatted: Bullets and Numbering

Formatted: Normal

B.1.3 SCAClientFactoryFinder interface

The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can create alternative implementations of this interface that use different class loading or lookup mechanisms.

```
/*
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
 * OASIS trademark, IPR and other policies apply.
 */
```

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

import java.util.Properties;

/* A Service Provider Interface representing a SCAClientFactory finder.
 * SCA provides a default reference implementation of this interface.
 * SCA runtime vendors can create alternative implementations of this
 * interface that use different class loading or lookup mechanisms.
 */
public interface SCAClientFactoryFinder {

    SCAClientFactory find(Properties properties,
                          ClassLoader classLoader);
}
```

Formatted: Normal

Formatted: Bullets and Numbering

B.1.4 SCAClientFactoryFinderImpl class

This class provides a default implementation for finding a provider's SCAClientFactory implementation class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the base SCAClientFactory class.

It discovers a provider's SCAClientFactory implementation by referring to the following information in this order:

1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the newInstance() method call if specified
2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```
/*
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
 * OASIS trademark, IPR and other policies apply.
 */
package org.oasisopen.sca.client.impl;

import org.oasisopen.sca.client.SCAClientFactoryFinder;

import java.io.BufferedReader;
import java.io.Closeable;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.Properties;

import org.oasisopen.sca.SCARuntimeException;
import org.oasisopen.sca.client.SCAClientFactory;

/**
 * This is a default implementation of an SCAClientFactoryFinder which is
used
 * to find an implementation of the SCAClientFactory interface.
 *
 * @see SCAClientFactoryFinder
 * @see SCAClientFactory
 *
 * @author OASIS Open
 */
public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {
```

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

Formatted: French France

```
2731
2732        /**
2733         * The name of the System Property used to determine the SPI
2734         * implementation to use for the SCAClientFactory.
2735         */
2736        private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
2737          SCAClientFactory.class.getName();
2738
2739        /**
2740         * The name of the file loaded from the ClassPath to determine
2741         * the SPI implementation to use for the SCAClientFactory.
2742         */
2743        private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
2744            = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
2745
2746        /**
2747         * Public Constructor
2748         */
2749        public SCAClientFactoryFinderImpl() {
2750        }
2751
2752        /**
2753         * Creates an instance of the SCAClientFactorySPI implementation.
2754         * This discovers the SCAClientFactorySPI Implementation and instantiates
2755         * the provider's implementation.
2756         *
2757         * @param properties    Properties that may be used when creating a new
2758         * instance of the SCAClient
2759         * @param classLoader   ClassLoader that may be used when creating a new
2760         * instance of the SCAClient
2761         * @return new instance of the SCAClientFactorySPI
2762         * @throws SCARuntimeException Failed to create SCAClientFactorySPI
2763         * Implementation.
2764         */
2765        public SCAClientFactory find(Properties properties,
2766                                     ClassLoader classLoader) {
2767            if (classLoader == null) {
2768                classLoader = getThreadContextClassLoader();
2769            }
2770            final String factoryImplClassName =
2771                discoverProviderFactoryImplClass(properties, classLoader);
2772            final Class<? extends SCAClientFactory> factoryImplClass
2773                = loadProviderFactoryClass(factoryImplClassName,
2774    classLoader);
2775            final SCAClientFactory factory =
2776                instantiateSCAClientFactoryClass(factoryImplClass);
2777            return factory;
2778        }
2779
2780        /**
2781         * Gets the Context ClassLoader for the current Thread.
2782         *
2783         * @return The Context ClassLoader for the current Thread.
2784         */
2785        private static ClassLoader getThreadContextClassLoader() {
2786            final ClassLoader threadClassLoader =
2787                Thread.currentThread().getContextClassLoader();
2788            return threadClassLoader;
```

```
2789        }
2790
2791        /**
2792         * Attempts to discover the class name for the SCAClientFactorySPI
2793         * implementation from the specified Properties, the System Properties
2794         * or the specified ClassLoader.
2795         *
2796         * @return The class name of the SCAClientFactorySPI implementation
2797         * @throw SCARuntimeException Failed to find implementation for
2798         * SCAClientFactorySPI.
2799         */
2800        private static String
2801            discoverProviderFactoryImplClass(Properties properties,
2802                                             ClassLoader classLoader)
2803            throws SCARuntimeException {
2804            String providerClassName =
2805                checkPropertiesForSPIClassName(properties);
2806            if (providerClassName != null) {
2807                return providerClassName;
2808            }
2809
2810            providerClassName =
2811                checkPropertiesForSPIClassName(System.getProperties());
2812            if (providerClassName != null) {
2813                return providerClassName;
2814            }
2815
2816            providerClassName = checkMETAINFServicesForSIPClassName(classLoader);
2817            if (providerClassName == null) {
2818                throw new SCARuntimeException(
2819                      "Failed to find implementation for SCAClientFactory");
2820            }
2821
2822            return providerClassName;
2823        }
2824
2825        /**
2826         * Attempts to find the class name for the SCAClientFactorySPI
2827         * implementation from the specified Properties.
2828         *
2829         * @return The class name for the SCAClientFactorySPI implementation
2830         * or <code>null</code> if not found.
2831         */
2832        private static String
2833            checkPropertiesForSPIClassName(Properties properties) {
2834            if (properties == null) {
2835                return null;
2836            }
2837
2838            final String providerClassName =
2839                properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
2840            if (providerClassName != null && providerClassName.length() > 0) {
2841                return providerClassName;
2842            }
2843
2844            return null;
2845        }
2846
```

```
2847        /**
2848         * Attempts to find the class name for the SCAClientFactorySPI
2849         * implementation from the META-INF/services directory
2850         *
2851         * @return The class name for the SCAClientFactorySPI implementation or
2852         * <code>null</code> if not found.
2853         */
2854        private static String checkMETAINFServicesForSIPClassName(ClassLoader cl)
2855        {
2856            final URL url =
2857                cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
2858            if (url == null) {
2859                return null;
2860            }
2861
2862            InputStream in = null;
2863            try {
2864                in = url.openStream();
2865                BufferedReader reader = null;
2866                try {
2867                    reader =
2868                      new BufferedReader(new InputStreamReader(in, "UTF-8"));
2869
2870                    String line;
2871                    while ((line = readNextLine(reader)) != null) {
2872                        if (!line.startsWith("#") && line.length() > 0) {
2873                            return line;
2874                        }
2875                    }
2876
2877                    return null;
2878                } finally {
2879                    closeStream(reader);
2880                }
2881            } catch (IOException ex) {
2882                throw new SCARuntimeException(
2883                            "Failed to discover SCAClientFactory provider", ex);
2884            } finally {
2885                closeStream(in);
2886            }
2887        }
2888
2889        /**
2890         * Reads the next line from the reader and returns the trimmed version
2891         * of that line
2892         *
2893         * @param reader The reader from which to read the next line
2894         * @return The trimmed next line or <code>null</code> if the end of the
2895         * stream has been reached
2896         * @throws IOException I/O error occurred while reading from Reader
2897         */
2898        private static String readNextLine(BufferedReader reader)
2899            throws IOException {
2900
2901            String line = reader.readLine();
2902            if (line != null) {
2903                line = line.trim();
2904            }
```

```
2905            return line;
2906        }
2907
2908        /**
2909         * Loads the specified SCAClientFactory Implementation class.
2910         *
2911         * @param factoryImplClassName The name of the SCAClientFactory
2912         * Implementation class to load
2913         * @return The specified SCAClientFactory Implementation class
2914         * @throws SCARuntimeException Failed to load the SCAClientFactory
2915         * Implementation class
2916         */
2917        private static Class<? extends SCAClientFactory>
2918            loadProviderFactoryClass(String factoryImplClassName,
2919                                     ClassLoader classLoader)
2920            throws SCARuntimeException {
2921
2922            try {
2923                final Class<?> providerClass =
2924                      classLoader.loadClass(factoryImplClassName);
2925                final Class<? extends SCAClientFactory> providerFactoryClass =
2926                    providerClass.asSubclass(SCAClientFactory.class);
2927                return providerFactoryClass;
2928            } catch (ClassNotFoundException ex) {
2929                throw new SCARuntimeException(
2930                      "Failed to load SCAClientFactory implementation class "
2931                      + factoryImplClassName, ex);
2932            } catch (ClassCastException ex) {
2933                throw new SCARuntimeException(
2934                            "Loaded SCAClientFactory implementation class "
2935                           + factoryImplClassName
2936                      + " is not a subclass of "
2937                      + SCAClientFactory.class.getName() , ex);
2938            }
2939        }
2940
2941        /**
2942         * Instantiate an instance of the specified SCAClientFactorySPI
2943         * Implementation class.
2944         *
2945         * @param factoryImplClass The SCAClientFactorySPI Implementation
2946         * class to instantiate.
2947         * @return An instance of the SCAClientFactorySPI Implementation class
2948         * @throws SCARuntimeException Failed to instantiate the specified
2949         * specified SCAClientFactorySPI Implementation class
2950         */
2951        private static SCAClientFactory
2952            instantiateSCAClientFactoryClass(
2953                    Class<? extends SCAClientFactory> factoryImplClass)
2954            throws SCARuntimeException {
2955
2956            try {
2957                final SCAClientFactory provider = factoryImplClass.newInstance();
2958                return provider;
2959            } catch (Throwable ex) {
2960                throw new SCARuntimeException(
2961                   "Failed to instantiate SCAClientFactory implementation class "
2962                   + factoryImplClass, ex);
```

```
        }
    }

    /**
     * Utility method for closing Closeable Object.
     *
     * @param closeable The Object to close.
     */
    private static void closeStream(Closeable closeable) {
        if (closeable != null) {
            try{
                closeable.close();
            } catch (IOException ex) {
                throw new SCARuntimeException("Failed to close stream", ex);
            }
        }
    }
}
```

B.1.5 SCAClient Classes and Interfaces - what does a vendor need to do?

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

- Implement their SCAClientFactory and SCAClient implementation classes

  Vendors need to provide an implementation of SCAClient that is capable of looking up Services in their SCA Runtime.

  Vendors need to subclass SCAClientFactory and implement the createSCAClient() method so that it creates an instance of their SCAClient implementation.

- Configure the Vendor Implementation classes so they are used
  Vendors have several options:

  Option 1: Set System Property to point to the Vendor's implementation

  Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their implementation class and use the reference implementation of SCAClientFactoryFinder

  Option 2: Provide a META-INF/services file

  Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points to their implementation class and use the reference implementation of SCAClientFactoryFinder

  Option 3: Inject a vendor implementation instance into SCAClientFactory

  Vendors inject an instance into the defaultFactory field of SCAClientFactory. The SCAClientFactoryFinder is not used in this scenario.

Formatted: Bullets and Numbering

Deleted: Option 4: Provide a Vendor specific implementation of SCAClientFactoryFinder

Vendors write a new implementation of SCAClientFactoryFinder and replace the reference implementation that is provided by SCA.

C. Conformance Items

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

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of ***method overloading***.
[JCA20002]	the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
[JCA20003]	within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.
[JCA20004]	For a composite scope implementation instance, the SCA runtime MUST ensure that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.
[JCA20005]	When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.
[JCA20006]	If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.
[JCA20007]	the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.
[JCA30001]	The value of the @interface attribute MUST be the fully qualified name of the Java interface class
[JCA30002]	The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.
[JCA30004]	The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.
[JCA70001]	SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.
[JCA80001]	ComponentContext.getService method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..n or 1..n.

[JCA80002]	The ComponentContext.getRequestContext method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.
[JCA80003]	When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.
[JCA90001]	An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.
[JCA90002]	SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class.
[JCA90003]	If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation.
[JCA90004]	A method annotated with @Destroy MAY have any access modifier and MUST have a void return type and no arguments.
[JCA90005]	If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends.
[JCA90007]	When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started.
[JCA90008]	A method marked with the @Initi annotation MAY have any access modifier and MUST have a void return type and no arguments.
[JCA90009]	If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.
[JCA90011]	the @Property annotation MUST NOT be used on a class field that is declared as final.
[JCA90012]	the @Property annotation MUST be used in order to inject a property onto a non-public field.
[JCA90013]	For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present.

[JCA90014]	For a @Property annotation applied to a constructor parameter, the required attribute MUST have the value true.
[JCA90015]	The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
[JCA90016]	The @Reference annotation MUST NOT be used on a class field that is declared as final.
[JCA90017]	the @Reference annotation MUST be used in order to inject a reference onto a non-public field.
[JCA90018]	For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.
[JCA90019]	For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true.
[JCA90020]	If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation required attribute is true.
[JCA90021]	If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.
[JCA90022]	An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null (either via injection or via API call).
[JCA90023]	An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection (either via injection or via API call).
[JCA90024]	References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.
[JCA90025]	In order for reinjection to occur, the following MUST be true: 1. The component MUST NOT be STATELESS scoped. 2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed.
[JCA90026]	If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed.
[JCA90027]	If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException.

[JCA90028]	If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException.
[JCA90029]	If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.
[JCA90030]	A ServiceReference that has been obtained from a reference by ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast(). If the reference is subsequently reinjected, the ServiceReference obtained from the original reference MUST continue to work as if the reference target was not changed.
[JCA90031]	If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference.
[JCA90032]	If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.
[JCA90033]	If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.
[JCA90034]	A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place.
[JCA90035]	If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.
[JCA90036]	If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.
[JCA90037]	in the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring.
[JCA90038]	In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.
[JCA90039]	A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component.

Deleted: [JCA90031]
Deleted: [JCA90031]
Deleted: [JCA90032]
Deleted: [JCA90032]
Deleted: [JCA90033]
Deleted: [JCA90033]
Deleted: [JCA90034]
Deleted: [JCA90034]
Deleted: [JCA90035]
Deleted: [JCA90035]
Deleted: [JCA90036]
Deleted: [JCA90036]
Deleted: [JCA90037]
Deleted: [JCA90037]
Deleted: [JCA90038]
Deleted: [JCA90038]
Deleted: [JCA90039]
Deleted: [JCA90039]

[JCA90040]	The @Remotable annotation is used to specify a Java service interface as remotable. A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.
[JCA90041]	The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.
[JCA90042]	An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.
[JCA90043]	A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.
[JCA90044]	A @Service annotation with no attributes MUST be ignored, it is the same as not having the annotation there at all.
[JCA90045]	A component implementation MUST NOT have two services with the same Java simple name.
[JCA90046]	When used to annotate a method or a field of an implementation class for injection of a callback object, the@Callback annotation MUST NOT specify any attributes.
[JCA90047]	For a @Property annotation, if the the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false.
[JCA100001]	For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.
[JCA100002]	The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.
[JCA100003]	For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.
[JCA100004]	SCA runtimes MUST support the JAXB 2.1 mapping from Java types to XML schema types.
[JCA100005]	SCA runtimes MAY support the SDO 2.1 mapping from Java types to XML schema types.
[JCA100006]	For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.
[JCA100007]	For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.

Deleted: [JCA90045]

Deleted: [JCA90045]

[JCA100008]	If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation.
[JCA100009]	SCA runtimes MUST support the use of the JAX-WS client asynchronous model.

3018

D. 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]

E. Non-Normative Text

F. Revision History

[optional; should not be included in OASIS Standards]

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

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

3030

Page 54: [1] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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

Page 54: [2] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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

Page 54: [3] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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

Page 54: [4] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection

Page 54: [5] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection

Page 55: [6] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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.

Page 55: [7] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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.

Page 55: [8] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Page 55: [9] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Page 55: [10] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Page 55: [11] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Page 55: [12] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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.

Page 55: [13] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

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.

Page 55: [14] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.

Page 55: [15] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.

Page 55: [16] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.

Page 55: [17] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.

Page 55: [18] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.

Page 55: [19] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.