

# ***SCA Service Component Architecture***

## **Client and Implementation Model Specification for C++**

SCA Version 1.00, March 21 2007

Technical Contacts:	Andrew Borley	IBM Corporation
	David Haney	Rogue Wave Software
	Oisin Hurley	IONA Technologies
	Todd Little	BEA Systems, Inc.
	Brian Lorenz	Sybase, Inc.
	Conor Patten	IONA Technologies
	Pete Robbins	IBM Corporation
	Colin Thorne	IBM Corporation

## Copyright Notice

© Copyright BEA Systems, Inc., Cape Clear Software, International Business Machines Corp, Interface21, IONA Technologies, Oracle, Primeton Technologies, Progress Software, Red Hat, Rogue Wave Software, SAP AG., Siemens AG., Software AG., Sun Microsystems, Inc., Sybase Inc., TIBCO Software Inc., 2005, 2007. All rights reserved.

## License

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy, display and distribute the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:

1. A link or URL to the Service Component Architecture Specification at this location:
  - <http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications>
2. The full text of the copyright notice as shown in the Service Component Architecture Specification.

BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue Wave, SAP, Siemens AG., Software AG., Sun Microsystems, Sybase, TIBCO (collectively, the "Authors") agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE Service Components Architecture SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Component Architecture Specification or its contents without specific, written prior permission. Title to copyright in the Service Component Architecture Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

## **Status of this Document**

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

BEA is a registered trademark of BEA Systems, Inc.

Cape Clear is a registered trademark of Cape Clear Software

IONA and IONA Technologies are registered trademarks of IONA Technologies plc.

Oracle is a registered trademark of Oracle USA, Inc.

Progress is a registered trademark of Progress Software Corporation

Primeton is a registered trademark of Primeton Technologies, Ltd.

Red Hat is a registered trademark of Red Hat Inc.

Rogue Wave is a registered trademark of Quovadx, Inc

SAP is a registered trademark of SAP AG.

SIEMENS is a registered trademark of SIEMENS AG

Software AG is a registered trademark of Software AG

Sun and Sun Microsystems are registered trademarks of Sun Microsystems, Inc.

Sybase is a registered trademark of Sybase, Inc.

TIBCO is a registered trademark of TIBCO Software Inc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

**Table of Contents**

Copyright Notice .....	ii
License .....	ii
Status of this Document .....	iii
Table of Contents .....	iv
1. Client and Implementation Model .....	1
1.1. Introduction .....	1
1.1.1. Use of Annotations .....	1
1.2. Basic Component Implementation Model .....	1
1.2.1. Implementing a Service .....	1
1.2.2. Conversational and Non-Conversational services .....	5
1.2.3. Component Implementation Scopes .....	5
1.2.4. Implementing a Configuration Property .....	7
1.2.5. Component Type and Component .....	8
1.2.6. Instantiation .....	10
1.3. Basic Client Model .....	10
1.3.1. Accessing Services from Component Implementations .....	10
1.3.2. Accessing Services from non-SCA component implementations .....	11
1.3.3. Calling Service Methods .....	12
1.4. Error Handling .....	12
1.5. Conversational Services .....	13
1.5.1. Conversational Client .....	14
1.5.2. Conversational Service Provider .....	14
1.5.3. Methods that End the Conversation .....	16
1.5.4. Passing Conversational Services as Parameters .....	16
1.5.5. Conversation Lifetime Summary .....	17
1.5.6. Application Specified Conversation IDs .....	17
1.5.7. Accessing Conversation IDs from Clients .....	18
1.6. Asynchronous Programming .....	18
1.6.1. Non-blocking Calls .....	18
1.6.2. Callbacks .....	19
1.7. C++ API .....	23
1.7.1. Reference Counting Pointers .....	23
1.7.2. Component Context .....	24

1.7.3.	ServiceReference .....	25
1.7.4.	SCAException .....	26
1.7.5.	SCANullPointerException .....	27
1.7.6.	ServiceRuntimeException .....	27
1.7.7.	ServiceUnavailableException .....	27
1.7.8.	NoRegisteredCallbackException .....	27
1.7.9.	ConversationEndedException .....	28
1.7.10.	MultipleServicesException.....	28
1.8.	C++ Annotations .....	28
1.8.1.	Interface Header Annotations.....	29
1.8.2.	Implementation Header Annotations .....	31
1.9.	WSDL to C++ and C++ to WSDL Mapping.....	35
2.	Appendix 1 .....	36
2.1.	Packaging.....	36
2.1.1.	Composite Packaging.....	36
3.	Appendix 2 .....	38
3.1.	Types Supported in Service Interfaces .....	38
3.1.1.	Local service .....	38
3.1.2.	Remotable service.....	38
4.	Appendix 3 .....	39
4.1.	Restrictions on C++ header files .....	39
5.	Appendix 4 .....	40
5.1.	XML Schemas.....	40
5.1.1.	sca-interface-cpp.xsd.....	40
5.1.2.	sca-implementation-cpp.xsd .....	41
6.	Appendix 5 .....	43
6.1.	C++ to WSDL Mapping.....	43
6.2.	Parameter and Return Type mappings .....	44
6.2.1.	Built-in, STL and SDO type mappings.....	44
6.2.2.	Binary data mapping .....	47
6.2.3.	Array mapping.....	47
6.2.4.	Multi-dimensional array mapping.....	48
6.2.5.	Pointer/reference mapping .....	48
6.2.6.	STL container mapping .....	49
6.2.7.	Struct mapping .....	50

6.2.8.	Enum mapping .....	51
6.2.9.	Union mapping .....	52
6.2.10.	Typedef mapping .....	52
6.2.11.	Pre-processor mapping .....	52
6.2.12.	Nesting types .....	52
6.2.13.	SDO mapping .....	54
6.2.14.	void* mapping.....	54
6.2.15.	User-defined types (UDT) mapping.....	54
6.2.16.	Included or Inherited types.....	55
6.3.	Namespace mapping .....	55
6.4.	Class mapping.....	56
6.5.	Method mapping.....	59
6.5.1.	Default parameter value mapping.....	59
6.5.2.	Non-named parameters and the return type .....	60
6.5.3.	The void return type .....	60
6.5.4.	No Parameters Specified .....	62
6.5.5.	In/Out Parameters .....	62
6.5.6.	Public Methods.....	62
6.5.7.	Inherited Public Methods .....	63
6.5.8.	Protected/Private Methods.....	63
6.5.9.	Constructors/Destructors.....	63
6.5.10.	Overloaded Methods .....	63
6.5.11.	Operator overloading .....	63
6.5.12.	Exceptions .....	63
7.	References.....	64

---

# 1. Client and Implementation Model

---

## 1.1. Introduction

This document describes the SCA Client and Implementation Model for the C++ programming language.

The SCA C++ implementation model describes how to implement SCA components in C++. A component implementation itself can also be a client to other services provided by other components or external services. The document describes how a C++ implemented component gets access to services and calls their methods.

The document also explains how non-SCA C++ components can be clients to services provided by other components or external services. The document shows how those non-SCA C++ component implementations access services and call their methods.

### 1.1.1. Use of Annotations

This document defines interface and implementation meta-data using annotations. The annotations are defined as C++ comments in interface and implementation header files, for example:

```
// @Scope("stateless")
```

All meta-data that is represented by annotations can also be defined in .composite and .componentType side files as defined in the SCA Assembly Specification and the extensions defined in this specification. Component type information found in the component type file must be compatible with any specified annotation information.

## 1.2. Basic Component Implementation Model

This section describes how SCA components are implemented using the C++ programming language. It shows how a C++ implementation based component can implement a local or remotable service, and how the implementation can be made configurable through properties.

A component implementation can itself be a client of services. This aspect of a component implementation is described in the basic client model section.

### 1.2.1. Implementing a Service

A component implementation based on a C++ class (a *C++ implementation*) provides one or more services.

40

41 The services provided by the C++ implementation have an interface which is defined using the  
 42 declaration of a C++ abstract base class. An abstract base class is a class which has only pure  
 43 virtual methods. This is the service interface.

44

45 The C++ class based component implementation must implement the C++ abstract base class  
 46 that defines the service interface.

47

48 The abstract base class for the service interface could be generated from a WSDL portType using  
 49 the mapping defined in this specification.

50

51 The following snippets show the C++ service interface and the C++ implementation class of a  
 52 C++ implementation.

53

54 Service interface.

55

```
56 // LoanService interface
57 class LoanService {
58 public:
59     virtual bool approveLoan(unsigned long customerNumber,
60                             unsigned long loanAmount) = 0;
61 };
62
```

63

64 Implementation declaration header file.

```
65 class LoanServiceImpl : public LoanService
66 {
67 public:
68     LoanServiceImpl();
69     virtual ~LoanServiceImpl();
70
71     virtual bool approveLoan(unsigned long customerNumber,
72                             unsigned long loanAmount);
73 };
74
```

75

76 Implementation.

77

```
78 #include "LoanServiceImpl.h"
79
80 LoanServiceImpl::LoanServiceImpl()
81 {
82     ...
83 }
84 LoanServiceImpl::~LoanServiceImpl()
85 {
```



```

86     ...
87 }
88
89 bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
90                                 unsigned long loanAmount)
91 {
92     ...
93 }
94

```

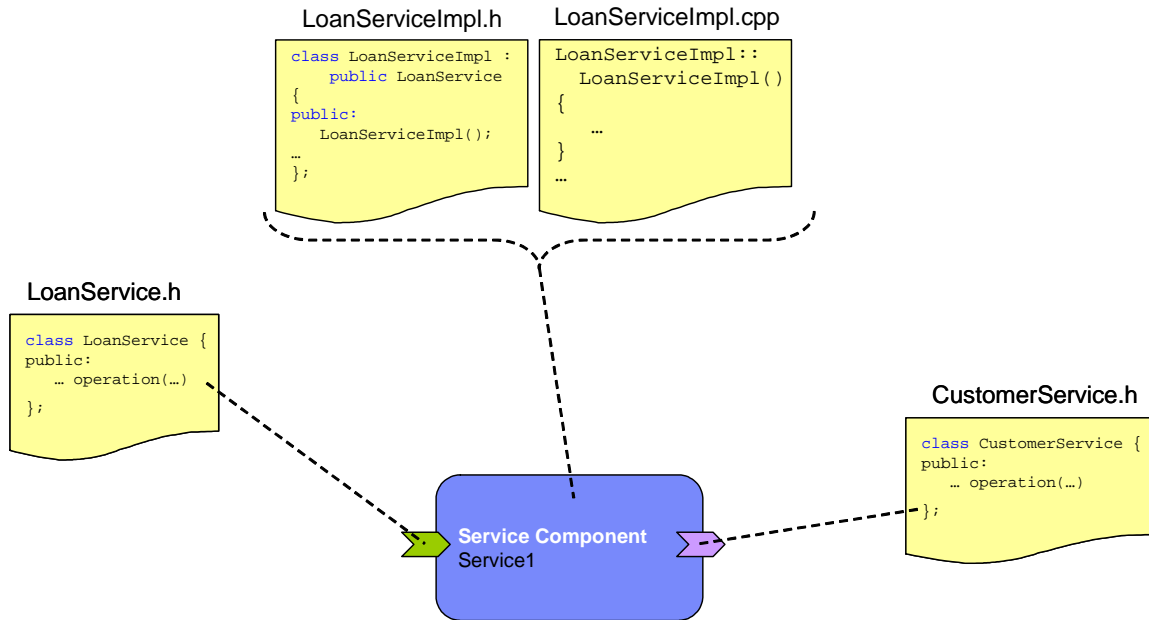
The following snippet shows the component type for this component implementation.

```

98 <?xml version="1.0" encoding="ASCII"?>
99 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
100     <service name="LoanService">
101         <interface.cpp header="LoanService.h"/>
102     </service>
103 </componentType>

```

The following picture shows the relationship between the C++ header files and implementation files for a component that has a single service and a single reference.



### 112 1.2.1.1. Implementing a Remotable Service

113 Remotable services are services that can be published through entry points. Published services  
 114 can be accessed by clients outside of the composite that contains the component that provides  
 115 the service.

116

117 Whether a service is remotable is defined by the interface of the service. This can be achieved by  
 118 using the **@Remotable** annotation in the C++ interface header or by adding the **remotable**  
 119 attribute to the C++ interface definition in the componentType file. The following snippet shows  
 120 the annotation of the interface header:

121

```
122 // LoanService interface
123 // @Remotable
124 class LoanService {
125 public:
126     virtual bool approveLoan(unsigned long customerNumber,
127                             unsigned long loanAmount) = 0;
128 };
129
```

130 The following snippet shows the component type for a remotable service:

```
131 <?xml version="1.0" encoding="ASCII"?>
132 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
133     <service name="LoanService">
134         <interface.cpp header="LoanService.h" remotable="true"/>
135     </service>
136 </componentType>
```

137

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

140

141 Complex data types exchanged via remotable service interfaces must be compatible with the  
 142 marshalling technology that is used by the service binding. For example, if the service is going  
 143 to be exposed using the standard web service binding, then the parameters must be Service  
 144 Data Objects (SDOs) [\[1\]](#).

145

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

148

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

153

154 An implementation of a remotable service can declare whether it allows pass by reference data  
 155 exchange semantics on calls to it, meaning that the by-value semantics can be maintained  
 156 without requiring that the parameters be copied. The implementation of remotable services that

allow pass by reference must not alter its input data during or after the invocation, and must not modify return data after invocation. The **@AllowsPassByReference** annotation on the implementation header of a remotable service is used to either declare that calls to the whole interface or individual methods allow pass by reference. Alternatively this can be specified in SCDL using the **allowsPassByReference=true** attribute on the implementation.cpp element or method definition.

#### 1.2.1.2. Implementing a Local Service

A local service can only be called by clients that are part of the same composite as the component that implements the local service. Local services cannot be published through entry points.

Whether a service is local is defined by its interface. In C++ this is indicated by the service not being defined as **remotable**.

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

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

#### 1.2.2. Conversational and Non-Conversational services

Service interfaces may be annotated to specify whether their contract is conversational, as described in the Assembly Specification [\[2\]](#) using the **@Conversation** annotation.

A non-conversational service, the default when no annotation is specified, indicates that the service contract is stateless between requests. A conversational service indicates that requests to the service are correlated.

#### 1.2.3. Component Implementation Scopes

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 the visibility and lifecycle contract an implementation has with the 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 scope.

Scopes may be specified using the **scope** attribute of the implementation.cpp or by specifying the **@Scope** annotation on the service class in the implementation header.

When a scope is not specified on an implementation class, the SCA runtime will interpret the implementation scope as **stateless**.

The SCA C++ Client and Implementation Model mandates support for the four basic scopes; **stateless**, **request**, **conversation**, and **composite**. Additional scopes may be provided by SCA runtimes.

The following snippet shows the component type for a composite scoped component:

```

204 <component name="LoanService">
205     <implementation.cpp library="loan" header="LoanServiceImpl.h"
206         scope="composite"/>
207 </component>

```

The following snippet shows how to define the implementation scope using the @Scope annotation:

```

211 // LoanServiceImpl header
212 // @Scope("composite")
213 class LoanServiceImpl : public LoanService
214 {
215     public:
216         LoanServiceImpl();
217         virtual ~LoanServiceImpl();
218
219         virtual bool approveLoan(unsigned long customerNumber,
220                                 unsigned long loanAmount);
221 };
222
223

```

For **stateless** scoped implementations, the SCA runtime will prevent concurrent execution of methods on an instance of that implementation. However, **composite** scoped implementations must be able to handle multiple threads running its methods concurrently.

The Following sections specify the mandatory scopes an SCA runtime must support for C++ component implementations.

### 1.2.3.1. Stateless scope

For stateless implementations, a different instance may be used to service each request. Implementation instances may be created or drawn from a pool of instances.

### 1.2.3.2. Request scope

Service requests are delegated to the same implementation instance for all collocated service invocations that occur while servicing a remote service request. The lifecycle of a request scope extends from the point a request on a remotable interface enters the SCA runtime and a thread processes that request until the thread completes synchronously processing the request.

There are times when a local request scoped service is called without a remotable service earlier in the call stack, such as when a local service is called from a non-SCA entity. In these cases, a

remote request is always considered to be present, but the lifetime of the request is implementation dependent. For example, a timer event could be treated as a remote request.

### 1.2.3.3. Composite scope

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

A composite scoped implementation may also specify eager initialization using the `@EagerInit` annotation or the `eagerInit=true` attribute on the `implementation.cpp` element of a component definition. When marked for eager initialization, the composite scoped instance will be created when its containing component is started.

### 1.2.3.4. Conversation scope

A conversation is defined as a series of correlated interactions between a client and a target service. A conversational scope starts when the first service request is dispatched to an implementation instance offering the target service. A conversational scope completes after an end operation defined by the service contract is called and completes processing or the conversation expires (see 1.5.3 Methods that End the Conversation). A conversation may be long-running and the SCA runtime may choose to passivate implementation instances. If this occurs, the runtime must guarantee implementation instance state is preserved.

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

## 1.2.4. Implementing a Configuration Property

Component implementations can be configured through properties. The properties and their types (not their values) are defined in the component type file or by using the `@Property` annotation on a data member of the implementation interface class. The C++ component can retrieve the properties using the `getProperties()` on the `ComponentContext` class.

The following code extract shows how to get the property values.

```

277     #include "ComponentContext.h"
278     using namespace osea::sca;
279
280     void clientMethod()
281     {
282         ...
283
284         ComponentContext context = ComponentContext::getCurrent();
285
286         DataObjectPtr properties = context.getProperties();
287
288         long loanRating = properties->getInteger("maxLoanValue");

```

```

289
290     ...
291 }
292

```

### 293 1.2.5. Component Type and Component

294 For a C++ component implementation, a component type must be specified in a side file. The  
 295 component type side file must be located in the same composite directory as the header file for  
 296 the implementation class.

297

298 This Client and Implementation Model for C++ extends the SCA Assembly model [\[2\]](#) providing  
 299 support for the C++ interface type system and support for the C++ implementation type.

300

301 The following snippet shows a partial schema for the C++ interface element used to type  
 302 services and references of component types. Additional attributes are described later in this  
 303 document.

304

```

305 <interface.cpp header="NCName" class="Name"? remotable="boolean"?
306     callbackHeader="NCName" callbackClass="Name"? />
307

```

308 The interface.cpp element has the following attributes:

- 309 • **header** – full name of the header file, including relative path from the composite root.  
 310 This header file describes the interface
- 311 • **class** – optional name of the class declaration in the header file, including any namespace  
 312 definition. If the header only contains one class then this class does not need to be  
 313 specified.
- 314 • **callbackHeader** – optional full name of the header file that describes the callback  
 315 interface, including relative path from the composite root.
- 316 • **callbackClass** – optional name of the class declaration for the call back in the callback  
 317 header file, including any namespace definition. If the header only contains one class then  
 318 this class does not need to be specified
- 319 • **remotable** – optional boolean value indicating whether the service is remotable or local.  
 320 The default is local.

321

322 The following snippet shows a partial schema for the C++ implementation element used to  
 323 define the implementation of a component. Additional attributes are described later in this  
 324 document.

325

```

326 <implementation.cpp library="NCName" path="NCName"? header="NCName" class="Name"?
327     scope="scope"? />
328

```

329 The implementation.cpp element has the following attributes:

- **library** – name of the dll or shared library that holds the factory for the service component. This is the root name of the library. On Windows the suffix “.dll” will be appended to this name. On linux the prefix “lib” and the suffix “.so” will be added.
- **path** - optional path to the library which is relative to the root of the composite.
- **header** – The name of the header file that declares the implementation class of the component. A path is optional which is relative to the root of the composite.
- **class** – optional name of the class declaration of the implementation, including any namespace definition. If the header only contains one class then this class does not need to be specified.
- **scope** – optional attribute indicating the scope of the component implementation. The default is stateless.

The following snippets show the C++ service interface and the C++ implementation class of a C++ service.

```

345 // LoanService interface
346 class LoanService {
347 public:
348     virtual bool approveLoan(unsigned long customerNumber,
349                             unsigned long loanAmount) = 0;
350 };
351

```

Implementation declaration header file.

```

353 class LoanServiceImpl : public LoanService
354 {
355 public:
356     LoanServiceImpl();
357     virtual ~LoanServiceImpl();
358
359     virtual bool approveLoan(unsigned long customerNumber,
360                             unsigned long loanAmount);
361 };
362
363

```

Implementation.

```

365 #include "LoanServiceImpl.h"
366
367 ///////////////////////////////////////////////////////////////////
368 // Construction/Destruction
369 ///////////////////////////////////////////////////////////////////
370 LoanServiceImpl::LoanServiceImpl()
371 {
372     ...
373 }
374
375 LoanServiceImpl::~LoanServiceImpl()
376 {
377     ...

```

```

378     }
379     ///////////////////////////////////////////////////////////////////
380     // Implementation
381     ///////////////////////////////////////////////////////////////////
382     bool LoanServiceImpl::approveLoan(unsigned long customerNumber,
383                                     unsigned long loanAmount)
384     {
385         ...
386     }
387

```

388 The following snippet shows the component type for this component implementation.

```

389
390     <?xml version="1.0" encoding="ASCII"?>
391     <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
392         <service name="LoanService">
393             <interface.cpp header="LoanService.h"/>
394         </service>
395     </componentType>
396

```

397 The following snippet shows the component using the implementation.

```

398
399     <?xml version="1.0" encoding="ASCII"?>
400     <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
401         name="LoanComposite" >
402         ...
403
404         <component name="LoanService">
405             <implementation.cpp library="loan" header="LoanServiceImpl.h"/>
406         </component>
407     </composite>
408

```

409

### 410 1.2.6. Instantiation

411 A C++ implementation class must provide a default constructor that can be used by a runtime to  
412 instantiate the component.

413

414

## 415 1.3. Basic Client Model

416 This section describes how to get access to SCA services from both SCA components and from  
417 non-SCA components. It also describes how to call methods of these services.

418

### 419 1.3.1. Accessing Services from Component Implementations

420 A component can get access to a service using a component context.

421

422 The following snippet shows the ComponentContext C++ class with its *getService()* method.



```

423 namespace osoa {
424     namespace sca {
425
426         class ComponentContext {
427         public:
428             static ComponentContextPtr getCurrent();
429             virtual void * getService(const std::string& referenceName) const = 0;
430             ...
431         }
432     }
433 }
434

```

435 The getService() method takes as its input argument the name of the reference and returns a  
436 pointer to an object providing access to the service. The returned object will implement the  
437 abstract base class definition that is used to describe the reference.

438

439 The following shows a sample of how the ComponentContext is used in a C++ component  
440 implementation. The getService() method is called on the ComponentContext passing the  
441 reference name as input. The return of the getService() method is cast to the abstract base class  
442 defined for the reference.

443

```

444 #include "ComponentContext.h"
445 #include "CustomerService.h"
446
447 using namespace osoa::sca;
448
449 void clientMethod()
450 {
451
452     unsigned long customerNumber = 1234;
453
454     ComponentContextPtr context = ComponentContext::getCurrent();
455
456     CustomerService* service =
457         (CustomerService* )context->getService("customerService");
458
459     short rating = service->getCreditRating(customerNumber);
460
461 }
462

```

463

### 464 1.3.2. Accessing Services from non-SCA component implementations

465

466 Non-SCA components can access component services by obtaining a ComponentContextPtr from  
467 the SCA runtime and then following the same steps as a component implementation as described  
468 above.

469

470 How an SCA runtime implementation allows access to and returns a ComponentContextPtr is not  
471 defined by this specification.

472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513

### 1.3.3. Calling Service Methods

The previous sections show the various options for getting access to a service. Once you have access to the service, calling a method of the service is like calling a method of a C++ class.

If you have access to a service whose interface is marked as remotable, then on calls to methods of that service you will experience remote semantics. Arguments and return are passed **by-value** and you may get a **ServiceUnavailableException**, which is a ServiceRuntimeException.

## 1.4. Error Handling

Clients calling service methods will experience business exceptions, and SCA runtime exceptions.

Business exceptions are raised by the implementation of the called service method. They should be caught by client invoking the method on the service.

SCA runtime exceptions are raised by the SCA runtime and signal problems in the management of the execution of components, and in the interaction with remote services. Currently the following SCA runtime exceptions are defined:

- **SCAException** – defines a root exception type from which all SCA defined exceptions derive.
  - **SCANullPointerException** – signals that code attempted to dereference a null pointer from a RefCountingPointer object.
  - **ServiceRuntimeException** - signals problems in the management of the execution of SCA components.
    - **ServiceUnavailableException** – signals problems in the interaction with remote services. This extends ServiceRuntimeException. These are exceptions that may be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since it most likely requires human intervention.
    - **MultipleServicesException** – signals that a method expecting identification of a single service is called where there are multiple services defined. Thrown by ComponentContext::getService(), ComponentContext::getSelfReference() and ComponentContext::getServiceReference().
    - **ConversationEndedException** – signals that a method has been called on a conversational service after the conversation was ended.
    - **NoRegisteredCallbackException** – signals that a callback was invoked on a service, but a callback method was not registered.

## 1.5. Conversational Services

A frequent pattern that occurs during the execution of remotable services is that a conversation is started between the client of the service and the provider of the service. The conversation is a series of method invocations that all pertain to a single common topic. For example, a conversation may be the series of service calls that are necessary in order to apply for a bank loan.

Conversations occur between a client and a target service. Consequently, requests originating from one client to multiple target services will result in multiple conversations. For example, if a client A calls B and C, implemented by conversational scoped classes, two conversations will result, one between A and B and another between A and C. Likewise, requests flowing through multiple implementation instances will result in multiple conversations. For example, a request originating from A to B to C will involve two conversations (A and B, B and C). In the previous example, if a request was then made from C to A, a third conversation would result (and the implementation instance for A would be different than the one making the original request).

Callback invocations will take place within the context of the conversation.

The mechanics for maintaining and flowing conversational ids remotely is delegated to the particular binding the request is made through.

For C++ component implementations, a service interface method may be decorated with the `@EndConversation` annotation to indicate to the SCA runtime that the current conversation should be ended when the method is invoked. A conversation may also be ended by calling the `endConversation()` method on a service reference.

The following is an example interface that has been annotated as being conversational:

```

// LoanService interface
// @Remotable
// @Scope("conversation")
class LoanService {
public:
    virtual void apply(LoanApplication application) = 0;
    virtual void lockCurrentRate(unsigned int termInYears) = 0;
    // @EndConversation
    virtual void cancelApplication( ) = 0;
    virtual int getLoanStatus( ) = 0;
};

```

The `cancelApplication()` method is annotated to end the conversation.

557 **1.5.1. Conversational Client**

558 There is no special coding required by the client of a conversational service. The developer of  
 559 the client knows that the service is conversational from the service interface definition. The  
 560 following shows an example client of the conversational service described above:

```
561
562     #include "LoanApplicationClientImpl.h"
563     #include "ComponentContext.h"
564     #include "LoanService.h"
565     #include "LoanApplication.h"
566
567     using namespace osea::sca;
568
569     void LoanApplicationClientImpl::clientMethod( LoanApplication loanApp,
570                                                  unsigned int term )
571     {
572         unsigned long customerNumber = 1234;
573
574         ComponentContextPtr context = ComponentContext::getCurrent();
575
576         // service is defined as member field: LoanService* service;
577         service = (LoanService* )context->getService("loanService");
578
579         service->apply( loanApp );
580         service->lockCurrentRate( term );
581     }
582
583
584     bool LoanApplicationClientImpl::isApproved()
585     {
586         return (service->getLoanStatus() == 1);
587     }
588
589
```

590 **1.5.2. Conversational Service Provider**

591

592 The provider of the conversational service also is not required to write special code to maintain  
 593 state if the implementation is annotated as conversation scoped.

594

595 There are a few capabilities that are available to the implementation of the service, but are not  
 596 required. The conversation ID can be retrieved from the ServiceReference:

```
597     ComponentContextPtr context = ComponentContext::getCurrent();
598     ServiceReferencePtr serviceRef = context->getSelfReference();
599     std::string conversationID = serviceRef->getConversationID();
600
```

601 The type of the conversation ID is a `std::string`. Application generated conversation IDs may be  
 602 other complex types, as described in the section below titled "Application Specified Conversation  
 603 IDs", that are serialized to a string.

604

605 The service implementation class may also have an optional `@Conversation` annotation that has  
 606 the following form:

```
607
608     @Conversation (maxIdleTime="10 minutes",
609                   maxAge="1 day",
610                   singlePrincipal=false)
```

611

612 The attributes of the `@Conversation` annotation have the following meaning:

- 613 • ***maxIdleTime*** - The maximum time that can pass between operations within a single  
 614 conversation. If more time than this passes, then the container may end the  
 615 conversation.
- 616 • ***maxAge*** - The maximum time that the entire conversation can remain active. If more  
 617 time than this passes, then the container may end the conversation.
- 618 • ***singlePrincipal*** – If true, only the principal (the user) that started the conversation has  
 619 authority to continue the operation.

620

621 Alternatively the conversation attributes can be specified on the implementation definition using  
 622 the **`conversationMaxIdleTime`**, **`conversationMaxAge`** and **`conversationSinglePrincipal`** of  
 623 `implementation.cpp`.

624

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

628

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

631

632 Here is an example implementation header file of a conversational service (the implementation  
 633 file is not shown).

634

```
635     // @Conversation(maxAge="30 days")
636     class LoanServiceImpl : public LoanService
637     {
638     public:
639         virtual void apply(LoanApplication application);
640         virtual void lockCurrentRate(unsigned int termInYears);
641         virtual void cancelApplication(void);
642         virtual int getLoanStatus();
643     };
644
```

645 The Conversation attributes may also be specified in the `<implementation.cpp>` element of a  
 646 component definition:

647

```

648 <component name="LoanService">
649     <implementation.cpp library="loan" header="LoanServiceImpl.h"
650         conversationMaxAge="30 days" />
651 </component>
652
653

```

### 654 1.5.3. Methods that End the Conversation

655 A method of a conversational service interface may be marked with an `@EndConversation`  
656 annotation. This means that once this method has been called, no further methods may be  
657 called so both the client and the target may free up resources that were associated with the  
658 conversation. It is also possible to mark a method on a callback interface (described later) as  
659 `@EndConversation`, in order for the service provider to be the party that chooses to end the  
660 conversation. If a method is called after the conversation completes a  
661 `ConversationEndedException` (which extends `ServiceRuntimeException`) is thrown. This may also  
662 occur if there is a race condition between the client and the service provider calling their  
663 respective `@EndConversation` methods.

664  
665 Alternatively the `endConversation="true"` attribute can be specified on a method in the  
666 `interface.cpp` element of a service.

### 669 1.5.4. Passing Conversational Services as Parameters

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

673 A service provider may also create a service reference for itself that it can pass to other services.  
674 A service implementation does this with a call to

```
675
676     ComponentContext::getSelfReference()
```

677  
678 or

```
679
680     ComponentContext::getSelfReference(const std::string& serviceName)
```

681  
682  
683 The second variant, which takes a `serviceName` parameter, must be used if the component  
684 implements multiple services.

685  
686 This may be used to support complex callback patterns, such as when a callback is applicable  
687 only to a subset of a larger conversation. Simple callback patterns are handled by the built-in  
688 callback support described later.

### 1.5.5. Conversation Lifetime Summary

#### Starting conversations

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

- A service is located using `ComponentContext::getService()` or `ComponentContext::getServices()`.
- A service reference is obtained using `ComponentContext::getServiceReference()` or `ComponentContext::getServiceReferences()`.

#### Continuing conversations

The client can continue an existing conversation, by:

- Holding the service reference that was created when the conversation started.
- Getting the service reference object passed as a parameter from another service, even remotely.

#### Ending conversations

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

- A service operation that has been annotated `@EndConversation` has been called.
- The service calls an `@EndConversation` method on a callback interface.
- The service's conversation lifetime timeout occurs.
- The client calls `ServiceReference::endConversation()`.
- The client calls `ServiceReference::setConversationID()` which implicitly ends any active conversation.

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

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

### 1.5.6. Application Specified Conversation IDs

It is also possible to take advantage of the state management aspects of conversational services while using a client-provided conversation ID. To do this, the client would use the `ServiceReference::setConversationID()` method.

```

729     ComponentContextPtr ctx = ComponentContext::getCurrent();
730
731     std::string conversationID("myID");
732
733     ServiceReferencePtr serviceReference = ctx->getServiceReference("loanService");
734     serviceReference->setConversationID(conversationID);
735
736     LoanService* service = (LoanService*)serviceReference->getService();
737
738

```

739 The conversation ID that is passed into this method should be an instance of a `std::string`. The  
740 ID must be unique to the client component over all time. If the client is not an SCA component,  
741 then the ID must be globally unique.

### 744 1.5.7. Accessing Conversation IDs from Clients

746 Whether the conversation ID is chosen by the client or is generated by the system, the client  
747 may access the conversation ID of a conversation by calling the  
748 `ServiceReference::getConversationID()` method on the service reference for the  
749 conversation.

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

## 756 1.6. Asynchronous Programming

758 Asynchronous programming of a service is where a client invokes a service and carries on  
759 executing without waiting for the service to execute. Typically, the invoked service executes at  
760 some later time. Output from the invoked service, if any, must be fed back to the client through  
761 a separate mechanism, since no output is available at the point where the service is invoked.  
762 This is in contrast to the call-and-return style of synchronous programming, where the invoked  
763 service executes and returns any output to the client before the client continues. The SCA  
764 asynchronous programming model consists of support for non-blocking method calls, callbacks,  
765 and conversational services. Each of these topics is discussed in the following sections.

### 767 1.6.1. Non-blocking Calls

768 Non-blocking calls represent the simplest form of asynchronous programming, where the client  
769 of the service invokes the service and continues processing immediately, without waiting for the  
770 service to execute.



772 Any method that returns "void" and has no declared exceptions may be marked by using the  
 773 **@OneWay** annotation on the method in the interface header or by using the **oneWay="true"**  
 774 attribute in the interface definition of the service. This means that the method is non-blocking  
 775 and communication with the service provider may use a binding that buffers the requests and  
 776 sends it at some later time.

777  
 778 The following snippet shows an interface header for a service with the reportEvent() method  
 779 declared as a one-way method:

```
780
781 // LoanService interface
782 class LoanService {
783 public:
784     virtual bool approveLoan(unsigned long customerNumber,
785                             unsigned long loanAmount) = 0;
786
787     // @OneWay
788     virtual void reportEvent(int eventId) = 0;
789
790 };
791
```

792 The following snippet shows the component type for a service with the reportEvent() method  
 793 declared as a one-way method:

```
794
795 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
796     <service name="LoanService">
797         <interface.cpp header="LoanService.h">
798             <method name="reportEvent" oneWay="true" />
799         </interface.cpp>
800     </service>
801 </componentType>
```

802  
 803 SCA does not currently define a mechanism for making non-blocking calls to methods that return  
 804 values or are declared to throw exceptions. It is recommended that service designers define  
 805 one-way methods as often as possible, in order to give the greatest degree of binding flexibility  
 806 to deployers.

### 807 1.6.2. Callbacks

808  
 809  
 810 A callback service is a service that is used for asynchronous communication from a service  
 811 provider back to its client in contrast to the communication through return values from  
 812 synchronous operations. Callbacks are used by **bidirectional services**, which are services that  
 813 have two interfaces:

- 814 1. an interface for the provided service
- 815 2. an interface that must be provided by the client

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

820

821 A callback interface is declared by the **callbackHeader** and optionally **callbackClass** attributes  
 822 in the interface definition of the service. The following snippet shows the component type for a  
 823 service **MyService** with the interface defined in **MyService.h** and the interface for callbacks  
 824 defined in **MyServiceCallback.h**,

825

```
826 <componentType xmlns="http://www.oesa.org/xmlns/sca/1.0" >
827   <service name="MyService">
828     <interface.cpp header="MyService.h"
829       callbackHeader="MyServiceCallback.h"/>
830   </service>
831 </componentType>
```

832

833 Alternatively the callback can be specified in the interface header using the **@Callback**  
 834 annotation:

835 MyService.h file:

```
836 // MyService interface
837 // @Callback(header="MyServiceCallback.h")
838 class MyService {
839 public:
840   virtual void someMethod( unsigned int arg ) = 0;
841 };
842
```

843 MyServiceCallback.h file:

```
844 // MyServiceCallback interface
845 class MyServiceCallback {
846 public:
847   virtual void receiveResult( unsigned int result ) = 0;
848 };
849
```

### 850 1.6.2.1. Stateful Callbacks

851

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

854 A component gets access to the callback service by using the `getCallback()` method of the  
 855 `ServiceReference` (obtained from the `ComponentContext`).

856

857 The following is an example service implementation for the service and callback declared above.  
 858 When the `someMethod` has completed its processing it retrieves the callback service from the  
 859 `ServiceReference` and invokes a callback method.

860

```
861 #include "MyServiceImpl.h"
862 #include "MyServiceCallback.h"
```

```

863     #include "osoa/sca/ComponentContext.h"
864     using namespace osoa::sca;
865
866     MyServiceImpl::someMethod( unsigned int arg )
867     {
868         ...
869         // do some processing...
870
871         ComponentContextPtr context = ComponentContext::getCurrent();
872         ServiceReferencePtr serviceRef = context->getSelfReference();
873         MyServiceCallback* callback = (MyServiceCallback* ) serviceRef-
874 >getCallback();
875         callback->receiveResult(result);
876     }
877

```

878 The following shows how a client component would invoke the MyService service and receive  
879 the callback.

```

880
881     #include "MyServiceImpl.h"
882     #include "MyServiceCallback.h"
883     #include "osoa/sca/ComponentContext.h"
884     using namespace osoa::sca;
885
886     void clientMethod( unsigned int arg )
887     {
888         // locate the service
889         ComponentContextPtr context = ComponentContext::getCurrent();
890         MyService* service = (MyService*)context->getService("myservice");
891         service->someMethod(arg);
892     }
893
894     MyServiceCallback::receiveResult(unsigned int result)
895     {
896         // code to process result
897     }
898
899

```

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

908  
909

#### 910 1.6.2.2. *Stateless Callbacks*

911 A stateless callback interface is a callback whose interface is not *conversational*. Unlike stateful  
912 services, the client that uses stateless callbacks will not have callback methods routed to an

instance of the client that contains any state that is relevant to the conversation. As such, it is the responsibility of such a client to perform any persistent state management itself. The only information that the client has to work with (other than the parameters of the callback method) is a callback ID that is passed with requests to the service and is guaranteed to be returned with any callback. The callback ID is retrieved from the ServiceReference.

The following snippets show a client setting a callback id before invoking the asynchronous service and the callback method retrieving the callback ID:

```

void clientMethod( unsigned int arg )
{
    // locate the service
    ComponentContextPtr context = ComponentContext::getCurrent();
    ServiceReferencePtr svcRef = context->getServiceReference("myservice");
    svcRef->setCallbackID("1234");
    MyService* service = (MyService*)svcRef->getService();
    service->someMethod(arg);
}

MyServiceCallback::receiveResult(unsigned int result)
{
    ComponentContextPtr context = ComponentContext::getCurrent();

    ServiceReferencePtr serviceRef = context->getSelfReference();
    std::string id = serviceRef->getCallbackID();

    // code to process result
}

```

### 1.6.2.3. Implementing Multiple Bidirectional Interfaces

Since it is possible for a single class to implement multiple services, it is also possible for callbacks to be defined for each of the services that it implements. To access the callbacks the ServiceReference::getCallback(serviceName) method must be used, passing in the name of the service for which the callback is to be obtained.

### 1.6.2.4. Customizing the Callback

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

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

963  
 964 A callback object for a stateful callback interface has the additional requirement that it must be  
 965 serializable. The SCA runtime may serialize a callback object and persistently store it.

966  
 967 A callback object may be a service reference to another service. In that case, the callback  
 968 messages go directly to the service that has been set as the callback. If the callback object is  
 969 not a service reference, then callback messages go to the client and are then routed to the  
 970 specific instance that has been registered as the callback object. However, if the callback  
 971 interface has a stateless scope, then the callback object **must** be a service reference.

### 972 973 **1.6.2.5. Customizing the Callback Identity**

974  
 975 The identity that is used to identify a callback request is, by default, generated by the system.  
 976 However, it is possible to provide an application specified identity that should be used to identify  
 977 the callback by calling the `ServiceReference::setCallbackID()` method. This can be used in  
 978 either stateful or stateless callbacks. The identity will be sent to the service provider, and the  
 979 binding must guarantee that the service provider will send the ID back when any callback  
 980 method is invoked.

981  
 982 The callback identity has the same restrictions as the conversation ID. It must be a `std::string`.

## 983 984 **1.7. C++ API**

985  
 986 All the C++ interfaces are found in the namespace `osoa::sca`, which has been omitted from the  
 987 following descriptions for clarity.

### 988 989 **1.7.1. Reference Counting Pointers**

990 These are a derived version of the familiar smart-pointer. The pointer class holds a real (dumb)  
 991 pointer to the object. If the reference counting pointer is copied, then a duplicate pointer is  
 992 returned with the same real pointer. A reference count within the object is incremented for each  
 993 copy of the pointer, so only when all pointers go out of scope will the object be freed.

994 `RefCountingPointer` defines methods raw pointer like semantics. This includes defining operators  
 995 for dereferencing the pointer (`*`, `->`), as well as operators for determining the validity of the  
 996 pointer.

```
997 template <typename T>
998 class RefCountingPointer {
999 public:
1000     T& operator* () const;
1001     T* operator-> () const;
```

```

1002     operator void* () const;
1003     bool operator! () const;
1004 };
1005

```

1006 The RefCountingPointer class has the following methods:

- 1007 • **operator\*O** – Dereferences the underlying pointer, returning a reference to the value. This is equivalent to calling \*p where p is the underlying pointer. If this method is invoked on null pointer, an SCANullPointerException is thrown.
- 1010 • **operator->O** – Allows methods to be invoked directly on the underlying pointer. This is equivalent to invoking p->func() where func() is a method defined on the underlying pointer type. If this method is invoked on a null pointer, an SCANullPointerException is thrown.
- 1014 • **operator void\*O** – Returns null if the underlying pointer is null, otherwise returns a non-zero value. This method allow for checking whether a RefCountingPointer is set, i.e. if (p) { /\* do something \*/ }.
- 1017 • **operator!O** – Returns true if the underlying pointer is null, false otherwise. This method allows for checking wither is RefCountingPointer is not set, i.e. if (!p) { /\* do something \*/ }

1020

1021 Reference counting pointers in SCA have the same name as the type they are pointing to, with a  
 1022 suffix of Ptr. (E.g. ComponentContextPtr, ServiceReferencePtr).

1023

1024

### 1025 1.7.2. Component Context

1026 The following shows the ComponentContext interface.

1027

```

1028 class ComponentContext {
1029 public:
1030     static ComponentContextPtr getCurrent();
1031
1032     virtual std::string getURI() const = 0;
1033
1034     virtual void * getService(const std::string& referenceName) const = 0;
1035     virtual std::list<void*> getServices(const std::string& referenceName)
1036                                         const = 0;
1037
1038     virtual ServiceReferencePtr getServiceReference(const std::string&
1039                                                    referenceName) const = 0;
1040     virtual std::list<ServiceReferencePtr> getServiceReferences(const std::string&
1041                                                                referenceName) const = 0;
1042
1043
1044     virtual DataObjectPtr getProperties() const = 0;
1045     virtual DataFactoryPtr getDataFactory() const = 0;
1046
1047
1048     virtual ServiceReferencePtr getSelfReference() const = 0;
1049     virtual ServiceReferencePtr getSelfReference(const std::string& serviceName)

```

```

1050                                     const = 0;
1051 };
1052

```

1053 The ComponentContext C++ interface has the following methods:

- 1054 • **getCurrent()** – returns the ComponentContext for the current component
- 1055 • **getURI()** – returns the absolute URI of the component.
- 1056 • **getService()** – returns a pointer to object implementing the interface defined for the  
1057 named reference. Input to the method is the name of a reference defined on this  
1058 component. A MultipleServicesException will be thrown if the reference resolves to more  
1059 than one service.
- 1060 • **getServices()** – returns a list of objects implementing the interface defined for the  
1061 named reference. Input to the method is the name of a reference defined on this  
1062 component.
- 1063 • **getServiceReference()** – returns a ServiceReference for the specified service. A  
1064 MultipleServicesException will be thrown if the reference resolves to more than one  
1065 service.
- 1066 • **getServiceReferences()** – returns a list of ServiceReferences for the named reference.
- 1067 • **getProperties()** – Returns an SDO from which all the properties defined in the  
1068 componentType file can be retrieved.
- 1069 • **getDataFactory()** – Returns an SDO DataFactory which can be used to create data  
1070 objects.
- 1071 • **getSelfReference()** – Returns a ServiceReference that can be used to invoke this  
1072 component over the designated service. The second variant, which takes a serviceName  
1073 parameter, must be used if the component implements multiple services. A  
1074 MultipleServicesException is thrown if the first variant is called for a component  
1075 implementing multiple services.

1076

1077

### 1078 1.7.3. ServiceReference

1079 The following shows the ServiceReference interface.

1080

```

1081 class ServiceReference {
1082 public:
1083     virtual void* getService() const = 0;
1084
1085     virtual std::string getConversationID() const = 0;
1086     virtual void setConversationID(const std::string& id) = 0;
1087
1088     virtual std::string getCallbackID() const = 0;
1089     virtual void setCallbackID(const std::string& id) = 0;
1090
1091     virtual void* getCallback() const = 0;
1092     virtual void setCallback(void* callback) = 0;
1093
1094     virtual void endConversation() = 0;
1095

```

1096  
1097 };  
1098

1099 The ServiceReference interface has the following methods:

1100

- 1101 • **getService()** – returns a pointer to the service for this reference. A  
1102 MultipleServicesException will be thrown if the reference resolves to more than one  
1103 service.
- 1104 • **getConversationID()** – returns the conversation ID
- 1105 • **setConversationID()** – sets a user provided conversation ID
- 1106 • **getCallbackID()** – returns the callback ID
- 1107 • **setCallbackID()** – sets the callback ID
- 1108 • **getCallback()** – returns the callback service
- 1109 • **setCallback()** – sets the callback service
- 1110 • **endConversation()** – ends the conversation for this service reference

1111

1112 The detailed description of the usage of these methods is described in the sections on  
1113 Conversational Services and Asynchronous Programming in this document.

1114

#### 1115 1.7.4. SCAException

1116 The following shows the SCAException interface.

1117

```
1118 namespace osoa {
1119     namespace sca {
1120
1121         class SCAException : public std::exception {
1122         public:
1123             const char* getEClassName() const;
1124             const char* getMessageText() const;
1125             const char* getFileName() const;
1126             unsigned long getLineNumber() const;
1127             const char* getFunctionName() const;
1128         };
1129     }
1130 }
```

1131

1132 The SCAException C++ interface has the following methods:

- 1133 • **getEClassName()** – Returns the type of the exception as a string. e.g.  
1134 "ServiceUnavailableException".
- 1135 • **getMessageText()** – Returns the message which the SCA runtime attached to the  
1136 exception.
- 1137 • **getFileName()** – Returns the filename within which the exception occurred – May be  
1138 zero if the filename is not known.
- 1139 • **getLineNumber()** – Returns the line number at which the exception occurred.



- **`getFunctionName()`** – Returns the function name in which the exception occurred.

Details concerning this class and its derived types are described in the section on Error Handling in this document.

### 1.7.5. SCANullPointerException

The following shows the SCANullPointerException interface.

```

namespace osoa {
    namespace sca {

        class SCANullPointerException : public SCAException {
        };
    }
}

```

### 1.7.6. ServiceRuntimeException

The following shows the ServiceRuntimeException interface.

```

namespace osoa {
    namespace sca {

        class ServiceRuntimeException : public SCAException {
        };
    }
}

```

### 1.7.7. ServiceUnavailableException

The following shows the ServiceUnavailableException interface.

```

namespace osoa {
    namespace sca {

        class ServiceUnavailableException : public ServiceRuntimeException {
        };
    }
}

```

### 1.7.8. NoRegisteredCallbackException

The following shows the NoRegisteredCallbackException interface.

```

1180
1181     namespace osoa {
1182         namespace sca {
1183
1184             class NoRegisteredCallbackException : public ServiceRuntimeException {
1185                 };
1186         }
1187     }
1188
1189

```

### 1190 1.7.9. ConversationEndedException

1191 The following shows the ConversationEndedException interface.

```

1192
1193     namespace osoa {
1194         namespace sca {
1195
1196             class ConversationEndedException : public ServiceRuntimeException {
1197                 };
1198         }
1199     }
1200

```

### 1201 1.7.10. MultipleServicesException

1202 The following shows the MultipleServicesException interface.

```

1203
1204     namespace osoa {
1205         namespace sca {
1206
1207             class MultipleServicesException : public ServiceRuntimeException {
1208                 };
1209         }
1210     }
1211

```

1212

1213

## 1214 1.8. C++ Annotations

1215

1216 This section provides definitions of the annotations which can be used in the interface and  
1217 implementation headers. The annotations are defined as C++ comments in interface and  
1218 implementation header files, for example:

1219

```

1220     // @Scope("stateless")

```

1221

1222 All meta-data that is represented by annotations can also be defined in .composite and  
1223 .componentType side files as defined in the SCA Assembly Specification and the extensions  
1224 defined in this specification. Component type information found in the component type file must  
1225 be compatible with any specified annotation information.

1226

1227 **1.8.1. Interface Header Annotations**

1228

1229 This section lists the annotations that may be used in the header file that defines a service  
 1230 interface. These annotations can also be represented in SCDL within the <interface.cpp>  
 1231 element.

1232

1233 **1.8.1.1. @Remotable**

1234 Annotation on service interface class to indicate that a service is remotable.

1235 Format:

```
1236 // @Remotable
```

1237

1238 The default is **false** (not remotable).

1239

1240 Interface header:

```
1241 // @Remotable
1242 class LoanService {
1243     ...
1244 };
```

1245

1246 Service definition:

```
1247
1248 <service name="LoanService">
1249     <interface.cpp header="LoanService.h" remotable="true" />
1250 </service>
```

1251

1252

1253 **1.8.1.2. @Callback**

1254 Annotation on a service interface class to specify the callback interface.

1255 Format:

```
1256 // @Callback(header="headerName", class="className")
```

1257

1258 where **headerName** is the name of the header defining the callback service interface.

1259 **className** is the optional name of the class for the callback interface.

1260

1261 Interface header:

1262

```
1263 // @Callback(header="MyServiceCallback.h", class="MyServiceCallback")
1264 class MyService {
1265 public:
1266     virtual void someMethod( unsigned int arg ) = 0;
```

1267       };

1268

1269

1270       Service definition:

1271

```
1272       <service name="MyService">
1273           <interface.cpp header="MyService.h"
1274               callbackHeader="MyServiceCallback.h"
1275               callbackClass="MyServiceCallback" />
1276       </service>
```

1277

1278

### 1279   **1.8.1.3. @OneWay**

1280       Annotation on a service interface method to indicate the method is one way.

1281       Format:

```
1282       // @OneWay
```

1283

1284       Interface header:

1285

```
1286       class LoanService
1287       {
1288       public:
1289           // @OneWay
1290           virtual void reportEvent(int eventId) = 0;
1291           ...
1292       };
```

1293

1294       Service definition:

```
1295       <service name="LoanService">
1296           <interface.cpp header="LoanService.h">
1297               <method name="reportEvent" oneWay="true" />
1298           </interface.cpp>
1299       </service>
```

1301

### 1302   **1.8.1.4. @EndConversation**

1303       Annotation on a service interface method to indicate that the conversation will be ended when  
1304       this method is called

1305       Format:

```
1306       // @EndConversation
```

1307

1308       Interface header:

1309

```

1310     class LoanService
1311     {
1312     public:
1313         // @EndConversation
1314         virtual void cancelApplication( ) = 0;
1315         ...
1316     };
1317

```

1318 Service definition:

```

1319     <service name="LoanService">
1320         <interface.cpp header="LoanService.h">
1321             <method name=" cancelApplication" endConversation="true" />
1322         </interface.cpp>
1323     </service>
1324
1325

```

1326

1327

## 1328 1.8.2. Implementation Header Annotations

1329

1330 This section lists the annotations that may be used in the header file that defines a service  
 1331 implementation. These annotations can also be represented in SCDL within the  
 1332 <implementation.cpp> element.

1333

### 1334 1.8.2.1. @Scope

1335 Annotation on a service implementation class to indicate the scope of the service.

1336 Format:

```

1337     // @Scope("value")

```

1338

1339 where **value** can be **stateless**, **composite**, **request** or **conversation**. The default value is  
 1340 **stateless**.

1341

1342 Implementation header:

```

1343     // @Scope("composite")
1344     class LoanServiceImpl : public LoanService {
1345         ...
1346     };
1347

```

1348 Component definition:

1349

```

1350     <component name="LoanService">
1351         <implementation.cpp library="loan" header="LoanServiceImpl.h"
1352             scope="composite" />
1353     </component>

```

1354

1355 **1.8.2.2. @EagerInit**

1356 Annotation on a service implementation class to indicate the implantation is to be instantiated  
1357 when its containing component is started.

1358 Format:

1359 `// @EagerInit`

1360

1361 Implementation header:

```
1362 // @EagerInit
1363 class LoanServiceImpl : public LoanService {
1364     ...
1365 };
1366
```

1367 Component definition:

1368

```
1369 <component name="LoanService">
1370     <implementation.cpp library="loan" header="LoanServiceImpl.h"
1371         eagerInit="true" />
1372 </component>
1373
```

1374 **1.8.2.3. @AllowsPassByReference**

1375 Annotation on service implementation class or method to indicate that a service or method allows  
1376 pass by reference semantics.

1377 Format:

1378 `// @AllowsPassByReference`

1379

1380 Implementation header:

```
1381 // @AllowsPassByReference
1382 class LoanService {
1383     ...
1384 };
1385
```

1386 Component definition:

1387

```
1388 <component name="LoanService">
1389     <implementation.cpp library="loan" header="LoanServiceImpl.h"
1390         allowsPassByReference="true" />
1391 </component>
1392
1393
```

1394

1395 **1.8.2.4. @Conversation**

1396 Annotation on a service implementation class to specify attributes of a conversational service.

1397 Format:

```
1398 // @Conversation(maxIdleTime="value", maxAge="value", singlePrincipal=boolValue)
```

1399

1400 where **value** is a time expressed as an integer followed by a space and then one of the  
1401 following: "seconds", "minutes", "hours", "days" or "years".

1402

1403 Implementation header:

1404

```
1405 // @Conversation(maxAge="30 days", maxIdleTime="5 minutes",
1406 //             singlePrincipal=false)
1407 class LoanServiceImpl : public LoanService
1408 {
1409     ...
1410 };
1411
```

1412 Component definition:

1413

```
1414 <component name="LoanService">
1415     <implementation.cpp library="loan" header="LoanServiceImpl.h"
1416         conversationMaxAge="30 days" conversationMaxIdle="5 minutes"
1417         conversationSinglePrincipal="false" />
1418 </component>
```

1419

1420

#### 1421 1.8.2.5. @Property

1422 Annotation on a service implementation class data member to define a property of the service.

1423 Format:

```
1424 // @Property(name="propertyName", type="typeQName"
```

```
1425 //             default="defaultValue", required="true")
```

1426

1427 where

- 1428 • **name (optional)** specifies the name of the property. If name is not specified the  
1429 property name is taken from the name of the following data member.
- 1430 • **type (optional)** specifies the type of the property. If not specified the type of the  
1431 property is based on the C++ mapping of the type of the following data member to an  
1432 xsd type as defined in the appendix **C++ to WSDL Mapping**. If the data member is an  
1433 array, then the property is many-valued.
- 1434 • **required (optional)** specifies whether a value must be set in the component definition  
1435 for this property. Default is **false**
- 1436 • **default (optional)** specifies a default value and is only needed if **required** is **false**,

1437

1438 Implementation:

```

1439 // @Property(name="loanType", type="xsd:int")
1440 long loanType;
1441

```

1442 Component Type definition:

```

1443
1444 <componentType ... >
1445     <service ... />
1446     <property name="loanType" type="xsd:int" />
1447 </componentType>
1448
1449

```

#### 1450 1.8.2.6. @Reference

1451 Annotation on a service implementation class data member to define a reference of the service.

1452 Format:

```

1453 // @Reference(name="referenceName", interfaceHeader="LoanService.h",
1454 //           interfaceClass="LoanService", required="true")

```

1455

1456 where

- 1457 • **name (optional)** specifies the name of the reference. If name is not specified the  
1458 reference name is taken from the name of the following data member.
- 1459 • **interfaceHeader (required)** specifies the C++ header defining the interface for the  
1460 reference.
- 1461 • **interfaceClass (optional)** specifies the C++ class defining the interface for the  
1462 reference. If not specified the class is derived from the type of the annotated data  
1463 member.
- 1464 • **required (optional)** specifies whether a value must be set for this reference. Default is  
1465 **true**

1466

1467 If the annotated data member is a `std::list` then the implied component type has a reference  
1468 with a multiplicity of either 0..n or 1..n depending on the value of the @Reference **required**  
1469 attribute – 1..n applies if `required=true`. Otherwise a multiplicity of 0..1 or 1..1 is implied.

1470

1471 Implementation:

```

1472 // @Reference(interfaceHeader="LoanService.h" required="true")
1473 LoanService* loanService;
1474
1475 // @Reference(interfaceHeader="LoanService.h" required="false")
1476 std::list<LoanService*> loanServices;
1477

```

1478 Component Type definition:

1479

```

1480 <componentType ... >

```



```
1481     <service ... />
1482     <reference name="loanService" multiplicity="1..1">
1483         <interface.cpp header="LoanService.h" class="LoanService" />
1484     </reference>
1485     <reference name="loanServices" multiplicity="0..n">
1486         <interface.cpp header="LoanService.h" class="LoanService" />
1487     </reference>
1488
1489 </componentType>
```

## 1.9. WSDL to C++ and C++ to WSDL Mapping

The SCA Client and Implementation Model for C++ applies the **WSDL to C++** mapping rules as defined by the OMG **WSDL to C++ Mapping Specification** [\[3\]](#) and the C++ to WSDL mapping rules as defined in the appendix **C++ to WSDL Mapping**.

---

## 2. Appendix 1

---

### 2.1. Packaging

#### 2.1.1. Composite Packaging

The physical realization of an SCA composite is a folder in a file system containing at least one .composite file. The following shows the MyValueComposite just after creation in a file system.

```
MyValue/
  MyValue.composite
```

Besides the .composite file the composite contains artifacts that define the implementations of components, and that define the bindings of services and references. Examples of artifacts would be C++ header files, shared libraries (for example, dll), WSDL portType definitions, XML schemas, WSDL binding definitions, and so on. These artifacts can be contained in subfolders of the composite, whereby programmers have the freedom to construct a folder structure that best fits the needs of their project. The following shows the complete MyValue composite folder file structure in a file system.

```
MyValue/
  MyValue.composite
  bin/
    myvalue.dll
  services/myValue/
    MyValue.h
    MyValueImpl.h
    MyValueImpl.componentType
    MyValueService.wsdl
  services/customer/
    CustomerService.h
    CustomerServiceImpl.h
    Customer.h
  services/stockquote/
    StockQuoteService.h
    StockQuoteService.wsdl
```

Note that the folder structure is not architected, other than the .composite file must be at the root of the folder structure.

**Addressing of the resources** inside of the composite is done relative to the root of the composite (i.e. the location of the .composite file).

Shared libraries (including dlls) will be located as specified in the <implementation.cpp> element in the .composite file relative to the root of the composite.

XML definitions like XML schema complex types or WSDL portTypes are referenced by composite and component type files using URIs. These URIs consist of the namespace and local name of

1547 these XML definitions. The composite folder or one of its subfolders has to contain the XML  
1548 resources providing the XML definitions identified by these URI's.  
1549

---

## 3. Appendix 2

---

### 3.1. Types Supported in Service Interfaces

A service interface can support a restricted set of the types available to a C++ programmer. This section summarizes the valid types that can be used.

#### 3.1.1. Local service

For a local service the types that are supported are:

- Any of the C++ primitive types (for example, int, short, char). In this case the types will be passed by value as is normal for C++.
- Pointers to any of the C++ primitive types (for example, int \*, short \*, char \*).
- The const keyword can be used for any pointer to a C++ primitive type (for example const char \*). If this is used on a parameter then the destination may not change the value.
- C++ class. The class will be passed by value as is normal for C++.
- Pointer to a C++ class. A pointer will be passed to the destination which can then modify the original contents.
- DataObjectPtr. An SDO pointer. This will be passed by reference.
- References to C++ classes (passed by reference)

#### 3.1.2. Remotable service

For a remotable service being called by another service the data exchange semantics is by-value. In this case the types that are supported are:

- Any of the C++ primitive types (for example, int, short, char). This will be copied.
- C++ classes. These will be passed using the copy constructor. The copy constructor must make sure that any embedded references, pointers or objects are copied appropriately.
- DataObjectPtr. An SDO pointer. The SDO will be copied and passed to the destination.

Not supported:

- Pointers.
- References.

---

## 4. Appendix 3

---

### 4.1. Restrictions on C++ header files

A C++ header file that is used to describe an interface has some restrictions. It must:

- Declare at least one class with:
  - At least one public method.
  - All public methods must be pure virtual (virtual with no implementation)

The following C++ keywords and constructs must not be used:

- Macros
- inline methods
- friend classes

---

## 5. Appendix 4

---

### 5.1. XML Schemas

Two new XML schemas are defined to support the use of C++ for implementation and definition of interfaces.

#### 5.1.1. sca-interface-cpp.xsd

```

1597 <?xml version="1.0" encoding="UTF-8"?>
1598 <schema xmlns="http://www.w3.org/2001/XMLSchema"
1599         targetNamespace="http://www.commonj.org/xmlns/sca/1.0/"
1600         xmlns:sca="http://www.commonj.org/xmlns/sca/1.0/"
1601         xmlns:sdo="commonj.sdo/XML"
1602         elementFormDefault="qualified">
1603
1604     <include schemaLocation="sca-core.xsd"/>
1605
1606     <element name="interface.cpp" type="sca:CPPInterface"
1607             substitutionGroup="sca:interface"/>
1608
1609     <complexType name="CPPInterface">
1610         <complexContent>
1611             <extension base="sca:Interface">
1612                 <sequence>
1613                     <element name="method" type="sca:CPPMethod"
1614                             minOccurs="0" maxOccurs="unbounded" />
1615                     <any namespace="##other" processContents="lax"
1616                             minOccurs="0" maxOccurs="unbounded"/>
1617                 </sequence>
1618                 <attribute name="header" type="NCName" use="required"/>
1619                 <attribute name="class" type="Name" use="required"/>
1620                 <attribute name="callbackHeader" type="NCName"
1621                             use="optional"/>
1622                 <attribute name="callbackClass" type="Name" use="optional"/>
1623                 <attribute name="remotable" type="boolean" use="optional"/>
1624                 <anyAttribute namespace="##any" processContents="lax"/>
1625             </extension>
1626         </complexContent>
1627     </complexType>
1628
1629     <complexType name="CPPMethod">
1630         <complexContent>
1631             <attribute name="name" type="NCName" use="required"/>
1632             <attribute name="oneWay" type="boolean" use="optional"/>
1633             <attribute name="endConversation" type="boolean" use="optional"/>
1634             <anyAttribute namespace="##any" processContents="lax"/>
1635         </complexContent>
1636     </complexType>
1637
1638 </schema>

```

## 5.1.2. sca-implementation-cpp.xsd

```

1639
1640 <?xml version="1.0" encoding="UTF-8"?>
1641 <schema xmlns="http://www.w3.org/2001/XMLSchema"
1642         targetNamespace="http://www.osea.org/xmlns/sca/1.0"
1643         xmlns:sca="http://www.osea.org/xmlns/sca/1.0"
1644         xmlns:sdo="commonj.sdo/XML"
1645         elementFormDefault="qualified">
1646
1647     <include schemaLocation="sca-core.xsd"/>
1648
1649     <element name="implementation.cpp" type="sca:CPPImplementation"
1650           substitutionGroup="sca:implementation" />
1651     <complexType name="CPPImplementation">
1652         <complexContent>
1653             <extension base="sca:Implementation">
1654                 <sequence>
1655                     <element name="method"
1656                           type="sca:CPPImplementationMethod"
1657                           minOccurs="0" maxOccurs="unbounded" />
1658                     <any namespace="##other" processContents="lax"
1659                           minOccurs="0" maxOccurs="unbounded" />
1660                 </sequence>
1661                 <attribute name="library" type="NCName" use="required"/>
1662                 <attribute name="header" type="NCName" use="required"/>
1663                 <attribute name="path" type="NCName" use="optional"/>
1664                 <attribute name="class" type="Name" use="optional"/>
1665                 <attribute name="scope" type="sca:CPPImplementationScope"
1666                           use="optional"/>
1667                 <attribute name="eagerInit" type="boolean" use="optional"/>
1668                 <attribute name="allowsPassByReference" type="boolean"
1669                           use="optional"/>
1670                 <attribute name="conversationMaxAge" type="string"
1671                           use="optional"/>
1672                 <attribute name="conversationMaxIdle" type="string"
1673                           use="optional"/>
1674                 <attribute name="conversationSinglePrincipal" type="boolean"
1675                           use="optional"/>
1676                 <anyAttribute namespace="##any" processContents="lax"/>
1677             </extension>
1678         </complexContent>
1679     </complexType>
1680
1681     <simpleType name="CPPImplementationScope">
1682         <restriction base="string">
1683             <enumeration value="stateless"/>
1684             <enumeration value="composite"/>
1685             <enumeration value="request"/>
1686             <enumeration value="conversion"/>
1687         </restriction>
1688     </simpleType>
1689
1690     <complexType name="CPPImplementationMethod">
1691         <complexContent>
1692             <attribute name="name" type="NCName" use="required"/>
1693             <attribute name="allowsPassByReference" type="boolean"
1694                       use="optional"/>

```

```
1695         <anyAttribute namespace="##any" processContents="lax"/>
1696     </complexContent>
1697 </complexType>
1698
1699 </schema>
1700
1701
```



---

## 6. Appendix 5

---

### 6.1. C++ to WSDL Mapping

This section describes a mapping from C++ header interfaces to a WSDL description of that interface. The intent is for implementations of this proposal to be able to deploy a service based only on a C++ header definition and for a WSDL definition of that service to be generated from the C++, either at deploy or run time.

This mapping currently only deals with producing document/literal wrapped style services and WSDL from C++ header files.

## 6.2. Parameter and Return Type mappings

This section details how types used as parameters or return types in C++ method prototypes get mapped to XML schema elements in the generated WSDL.

### 6.2.1. Built-in, STL and SDO type mappings

<i>C++ built in, STL and SDO types</i>	<i>Notes</i>	<i>XML Type</i>
bool		xsd:boolean
char	signed 8-bit <sup>1</sup>	xsd:byte
unsigned char	unsigned 8-bit <sup>1</sup>	xsd:unsignedByte
short	signed 16-bit <sup>1</sup>	xsd:short
unsigned short	unsigned 16-bit <sup>1</sup>	xsd:unsignedShort

<sup>1</sup> The size of this type is not fixed according to the C++ standard. The size indicated is the minimum size required by the C++ specification.

int	signed 16-bit <sup>1</sup>	xsd:short
unsigned int	unsigned 16-bit <sup>1</sup>	xsd:unsignedShort
long	signed 32-bit <sup>1</sup>	xsd:int
unsigned long	unsigned 32-bit <sup>1</sup>	xsd:unsignedInt
long long	signed 64-bit <sup>1</sup>	xsd:long
unsigned long long	unsigned 64-bit <sup>1</sup>	xsd:unsignedLong
float	32-bit floating point (IEEE-754-1985) <sup>1</sup>	xsd:float
double	64-bit floating point (IEEE-754-1985) <sup>1</sup>	xsd:double
long double	64-bit floating point (platform dependent, IEEE-754-1985) <sup>1</sup>	xsd:double
char* or char array	Must be a null-terminated UTF-	xsd:string

	8 encoded string	
wchar_t* or wchar_t array <sup>2</sup>	Must be a null-terminated UTF-16 or UTF-32 encoded string	xsd:string
std::string	Must be a UTF-8 encoded string	xsd:string
std::wstring <sup>2</sup>	Must be a UTF-16 or UTF-32 encoded string	xsd:string
commonj::sdo::DataObjectPtr		xsd:any

1718

1719

1720

1721

For example, a C++ method prototype defined in a header such as:

---

<sup>2</sup> The encoding associated with a wchar\_t variable is determined by the size of the wchar\_t type. If wchar\_t is a 16-bit type, UTF-16 is used, otherwise UTF-32 is used.

1722 long myMethod(char\* name, int id, double value);

1723

1724 would generate a schema like:

1725

```
1726 <xsd:element name="myMethod">
1727   <xsd:complexType>
1728     <xsd:sequence>
1729       <xsd:element name="name" type="xsd:string"/>
1730       <xsd:element name="id" type="xsd:int"/>
1731       <xsd:element name="value" type="xsd:double"/>
1732     </xsd:sequence>
1733   </xsd:complexType>
1734 </xsd:element>
```

1735

```
1736 <xsd:element name="myMethodResponse">
1737   <xsd:complexType>
1738     <xsd:sequence>
1739       <xsd:element name="myMethodResponseData" type="xsd:int"/>
1740     </xsd:sequence>
1741   </xsd:complexType>
1742 </xsd:element>
```

### 1743 6.2.2. Binary data mapping

1744 Binary data, such as data passed via non-null-terminated char\* or char arrays, is not supported  
 1745 in this mapping. char\* and char array parameters and return types are always mapped to  
 1746 xsd:string, and must be null-terminated. This requirement also applies to wchar\_t\* and wchar\_t  
 1747 array parameters.

### 1748 6.2.3. Array mapping

1749 C++ arrays passed in or out of methods get mapped as normal elements but with multiplicity  
 1750 allowed via the minOccurs and maxOccurs attributes. E.g. a method prototype such as

1751

1752 long myMethod(char\* name, int idList[], double value);

1753

1754 would generate a schema like:

1755

```
1756 <xsd:element name="myMethod">
1757   <xsd:complexType>
1758     <xsd:sequence>
1759       <xsd:element name="name" type="xsd:string"/>
1760       <xsd:element name="idList" type="xsd:int"
1761         minOccurs="0" maxOccurs="unbounded"/>
1762       <xsd:element name="value" type="xsd:double"/>
```

```

1763     </xsd:sequence>
1764 </xsd:complexType>
1765 </xsd:element>

```

#### 6.2.4. Multi-dimensional array mapping

Multi-dimensional arrays will need converting into nested elements. E.g. a method prototype such as

```

1769
1770 long myMethod(int multiIdArray[] [4] [2]);
1771

```

would generate a schema like:

```

1773
1774 <xsd:element name="myMethod">
1775   <xsd:complexType>
1776     <xsd:sequence>
1777       <xsd:element name="multiIdArray"
1778         minOccurs="0" maxOccurs="unbounded"/>
1779       <xsd:complexType>
1780         <xsd:sequence>
1781           <xsd:element name="multiIdArray"
1782             minOccurs="4" maxOccurs="4"/>
1783           <xsd:complexType>
1784             <xsd:sequence>
1785               <xsd:element name="multiIdArray" type="xsd:int"
1786                 minOccurs="2" maxOccurs="2" />
1787             </xsd:sequence>
1788           </xsd:complexType>
1789         </xsd:element>
1790       </xsd:sequence>
1791     </xsd:complexType>
1792   </xsd:element>
1793 </xsd:sequence>
1794 </xsd:complexType>
1795 </xsd:element>

```

#### 6.2.5. Pointer/reference mapping

A C++ method prototype that uses the 'pass-by-reference' style, defining parameters that are either references or pointers, is not meaningful when applied to web services, which rely on serialized data. A C++ method prototype that uses references or pointers will be converted to a WSDL operation that is defined as if the parameters were 'pass-by-value', with the web-service implementation framework responsible for creating the value, obtaining its pointer and passing that to the implementation class.

E.g. a C++ method prototype defined in a header such as:

```

1804

```

1805 `long myMethod(char* name, int* id, double* value);`

1806

1807 would generate a schema like:

1808

```
1809 <xsd:element name="myMethod">
1810   <xsd:complexType>
1811     <xsd:sequence>
1812       <xsd:element name="name" type="xsd:string"/>
1813       <xsd:element name="id" type="xsd:int"/>
1814       <xsd:element name="value" type="xsd:double"/>
1815     </xsd:sequence>
1816   </xsd:complexType>
1817 </xsd:element>
```

1818

1819 Note here how the `char*` type is a special case – `char*` parameters map to `xsd:string`.

1820 References and pointers are also used where in/out parameters are required – where the method  
1821 changes the value of the parameter and those changes are subsequently available in the  
1822 invoking code – see In/Out Parameters below.

### 1823 **6.2.6. STL container mapping**

1824 A C++ method prototype that uses STL containers (`std::vector`, `std::list`, `std::map`, `std::set`, etc)  
1825 as parameters or return types can be converted to a WSDL operation if the container is defined  
1826 as holding types that can be mapped. For example, a method such as:

1827

```
1828 std::string myMethod( DataMap myMap, int id );
```

1829

1830 with the `DataMap` type defined as an STL container holding mappable types:

1831

```
1832 typedef std::map<std::string, double> DataMap;
```

1833

1834 could convert to a schema like:

1835

```
1836 <xsd:element name="myMethod">
1837   <xsd:complexType>
1838     <xsd:sequence>
1839       <xsd:element name="myMap" type="DataMap"
1840         minOccurs="0" maxOccurs="unbounded"/>
1841       <xsd:element name="id" type="xsd:int"/>
1842     </xsd:sequence>
1843   </xsd:complexType>
1844 </xsd:element>
```

```

1845
1846 <xsd:complexType name="DataMap">
1847   <xsd:sequence>
1848     <xsd:element name="data1" type="xsd:string"/>
1849     <xsd:element name="data2" type="xsd:double"/>
1850   </xsd:sequence>
1851 </xsd:complexType>

```

### 6.2.7. Struct mapping

1853 C style structs that contain types that can be mapped, are themselves mapped to complex types.  
 1854 For example, a method such as:

```

1855
1856 std::string myMethod( DataStruct data, int id );

```

1857  
 1858 with the DataStruct type defined as a struct holding mappable types:

```

1859
1860 struct DataStruct {
1861   std::string name;
1862   double value;
1863 };
1864

```

1865 could convert to a schema like:

```

1866
1867 <xsd:element name="myMethod">
1868   <xsd:complexType>
1869     <xsd:sequence>
1870       <xsd:element name="data" type="DataStruct" />
1871       <xsd:element name="id" type="xsd:int"/>
1872     </xsd:sequence>
1873   </xsd:complexType>
1874 </xsd:element>
1875
1876 <xsd:complexType name="DataStruct">
1877   <xsd:sequence>
1878     <xsd:element name="name" type="xsd:string"/>
1879     <xsd:element name="value" type="xsd:double"/>
1880   </xsd:sequence>
1881 </xsd:complexType>

```

1882  
 1883 Handling of C++ style structs is not defined by this specification and is implementation  
 1884 dependent. In particular, C++ style structs that have protected or private data, or which require  
 1885 construction/destruction semantics may not be supported.



**6.2.8. Enum mapping**

In C++ enums define a list of named symbols that map to values. If a method uses an enum type, this can be mapped to a restricted element in the WSDL schema.

For example, a method such as:

```
std::string getValueFromType( ParameterType type );
```

with the ParameterType type defined as an enum:

```
enum ParameterType
{
    UNSET = 1,
    TYPEA,
    TYPEB,
    TYPEC
};
```

could convert to a schema like:

```
<xsd:element name="getValueFromType">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="type" type="ParameterType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="ParameterType">
  <xsd:restriction base="xsd:int">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="4"/>
  </xsd:restriction>
</xsd:simpleType>
```

The restriction used will have to be appropriate to the values of the enum elements. For example, a non-contiguous enum like:

```
enum ParameterType
{
    UNSET = 'u',
    TYPEA = 'A',
```

```

1927     TYPEB = 'B',
1928     TYPEC = 'C'
1929 };
1930

```

1931 could convert to a schema like:

```

1932
1933 <xsd:simpleType name="ParameterType">
1934   <xsd:restriction base="xsd:int">
1935     <xsd:enumeration value="86"/> <!-- Character 'u' -->
1936     <xsd:enumeration value="65"/> <!-- Character 'A' -->
1937     <xsd:enumeration value="66"/> <!-- Character 'B' -->
1938     <xsd:enumeration value="67"/> <!-- Character 'C' -->
1939   </xsd:restriction>
1940 </xsd:simpleType>

```

### 1941 6.2.9. Union mapping

1942 In C++ unions allow the same memory location to be used for different variables. Handling of  
 1943 C++ unions is not defined by this mapping, and is implementation dependent. For portability it  
 1944 is recommended that unions not be used in service interfaces.

### 1945 6.2.10. Typedef mapping

1946 Typedef mappings are supported by this specification, and will be followed when evaluating  
 1947 parameter and return types. This mapping does not define whether typedef names will be used  
 1948 in the resulting WSDL file. The use of these names is implementation dependent.

### 1949 6.2.11. Pre-processor mapping

1950 C++ allows for the use of pre-processor directives in order to control how a C++ header is  
 1951 parsed. Handling for pre-processor directives is not defined by this mapping, and support is  
 1952 implementation dependent. For portability it is recommended that pre-processor directives not  
 1953 be used in service interfaces.

### 1954 6.2.12. Nesting types

1955 If a struct, enum or STL container nests other structs, enums or STL containers within itself, it is  
 1956 mapped, as long as the nesting eventually boils down to a mappable type. For example, a  
 1957 method such as:

```

1958
1959 std::string myMethod(DataStruct data);

```

1960

1961 with types defined as follows:

```

1962
1963 struct DataStruct {
1964     std::string name;
1965     ValuesVector values;
1966     ParameterType type;
1967 };
1968

```

```

1969 typedef std::vector<double> ValuesVector;
1970
1971 enum ParameterType
1972 {
1973     UNSET = 1,
1974     TYPEA,
1975     TYPEB,
1976     TYPEC
1977 };
1978

```

1979 would convert to a schema like:

```

1980
1981 <xsd:element name="myMethod">
1982     <xsd:complexType>
1983         <xsd:sequence>
1984             <xsd:element name="data" type="DataStruct"/>
1985         </xsd:sequence>
1986     </xsd:complexType>
1987 </xsd:element>
1988
1989 <xsd:complexType name="DataStruct">
1990     <xsd:sequence>
1991         <xsd:element name="name" type="xsd:string"/>
1992         <xsd:element name="values" type="ValuesVector"/>
1993         <xsd:element name="type" type="ParameterType"/>
1994     </xsd:sequence>
1995 </xsd:complexType>
1996
1997 <xsd:complexType name="ValuesVector">
1998     <xsd:sequence>
1999         <xsd:element name="data" type="xsd:double"/>
2000     </xsd:sequence>
2001 </xsd:complexType>
2002
2003 <xsd:simpleType name="ParameterType">
2004     <xsd:restriction base="xsd:int">
2005         <xs:minInclusive value="1"/>
2006         <xs:maxInclusive value="4"/>
2007     </xsd:restriction>
2008 </xsd:simpleType>

```

**6.2.13. SDO mapping**

C++ method prototypes that use `commonj::sdo::DataObjectPtr` objects as parameter or return types are mapped to the any type in the WSDL schema as the schema for a Data Object is unknown before runtime. For example, a C++ method prototype defined in a header such as:

```
long myMethod(commonj::sdo::DataObjectPtr data);
```

would generate a schema like:

```
<xsd:element name="myMethod">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="data">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:any processContents="skip"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Typed (static) Data Objects are supported via the rules for User-defined types mapping below.

**6.2.14. void\* mapping**

The `void*` type is not supported due to its undefined nature.

**6.2.15. User-defined types (UDT) mapping**

C++ method prototypes that employ user-defined C++ types as return types or parameters are mapped if the C++ object defines setter and getter methods for its member variables. The types of the member variables must be mappable to a schema element via the rules in this document. The names of the schema elements are defined by the `set[Name]` and `get[Name]` methods. For example, a C++ method prototype defined in a header such as:

```
long myMethod(AnObject data);
```

where `AnObject` is defined in a locatable C++ header as:

```
class AnObject
{
public:
  AnObject();
```

```

2052     std::string getMyString() const;
2053     double getMyDouble() const;
2054
2055     void setMyString(std::string data);
2056     void setMyDouble(double otherData);
2057 };

```

2058

2059 would generate a schema like:

2060

```

2061 <xsd:element name="myMethod">
2062   <xsd:complexType>
2063     <xsd:sequence>
2064       <xsd:element name="data" type="AnObject"/>
2065     </xsd:sequence>
2066   </xsd:complexType>
2067 </xsd:element>

```

2068

```

2069 <xsd:complexType name="AnObject">
2070   <xsd:sequence>
2071     <xsd:element name="MyString" type="xsd:string"/>
2072     <xsd:element name="MyDouble" type="xsd:double"/>
2073   </xsd:sequence>
2074 </xsd:complexType>

```

2075

2076 Both set[Name] and get[Name] must be present in order for the variable to be mapped for the  
 2077 UDT type. In addition, any UDT must provide a default constructor.

2078

2079 This specification does not define support for arrays within UDTs. Instead it is recommended  
 2080 that classes use STL containers to represent collections.

### 2081 **6.2.16. Included or Inherited types**

2082 All types (structs, enums, classes, etc) that need to be mapped to WSDL schema must be able to  
 2083 be found from the C++ header being mapped. Implementations should allow a list of "include"  
 2084 directories to be specified. Types that are included (via a #include "SomeHeader.h" statement)  
 2085 or inherited from a superclass must be mappable to a schema element via the rules in this  
 2086 document.

2087

## 2088 **6.3. Namespace mapping**

2089 Where a C++ header defines a namespaced class, the namespace and class name should map to  
 2090 a target namespace used in the generated WSDL. For example, a header file such as:

2091

```

2092 namespace myCorp
2093 {
2094     namespace myServices

```

```

2095     {
2096         class ExampleService
2097         {
2098             public:
2099                 // Methods go here
2100             };
2101         }
2102     }
2103 
```

2104 would generate WSDL like:

```

2105
2106 <definitions name="ExampleService"
2107     xmlns="http://schemas.xmlsoap.org/wsd/"
2108     targetNamespace="http://myCorp/myServices/ExampleService"
2109     xmlns:soap="http://schemas.xmlsoap.org/wsd/soap/"
2110     xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
2111     <types>
2112         <xsd:schema
2113             targetNamespace="http://myCorp/myServices/ExampleService"
2114             xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
2115     ...
2116 
```

2117 Implementations should allow namespace mappings to be specified separately to override this  
2118 default behaviour.

2119

## 2120 **6.4. Class mapping**

2121 A single class in a C++ header maps to a single WSDL service element, a single WSDL binding  
2122 element and a single WSDL portType element. The WSDL service element contains a single  
2123 WSDL port element. The WSDL binding and WSDL portType elements each contain multiple  
2124 WSDL operation elements that map to the public methods defined in the C++ class. A pair of  
2125 WSDL message elements and a pair of XML schema elements are generated for each WSDL  
2126 operation. SOAP 1.1 binding and address information is also generated. For example, a C++  
2127 header such as:

```

2128
2129 class MyService
2130 {
2131     public:
2132         int myMethod(std::string data);
2133         double myOtherMethod(double otherData);
2134 };
2135 
```

2135

2136 would generate WSDL like:

```

2137
2138 <?xml version="1.0" encoding="UTF-8"?>
2139 <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2140     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
2141     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2142     xmlns:tns="http://MyService"
2143     targetNamespace="http://MyService">
2144     <types>
2145         <xsd:schema targetNamespace="http://sample/MyService"
2146             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2147             xmlns:tns="http://MyService" elementFormDefault="qualified">
2148
2149             <xsd:element name="myMethod">
2150                 <xsd:complexType>
2151                     <xsd:sequence>
2152                         <xsd:element name="data" type="xsd:string"/>
2153                     </xsd:complexType>
2154                 </xsd:element>
2155                 <xsd:element name="myMethodResponse">
2156                     <xsd:complexType>
2157                         <xsd:sequence>
2158                             <xsd:element name="myMethodResponseData" type="xsd:int"/>
2159                         </xsd:complexType>
2160                     </xsd:element>
2161
2162                 <xsd:element name="myOtherMethod">
2163                     <xsd:complexType>
2164                         <xsd:sequence>
2165                             <xsd:element name="otherData" type="xsd:double"/>
2166                         </xsd:sequence>
2167                     </xsd:complexType>
2168                 </xsd:element>
2169                 <xsd:element name="myOtherMethodResponse">
2170                     <xsd:complexType>
2171                         <xsd:sequence>
2172                             <xsd:element name="myMethodResponseData" type="xsd:double"/>
2173                         </xsd:complexType>
2174                     </xsd:element>
2175
2176                 </xsd:schema>
2177     </types>
2178

```

```

2179 <message name="myMethodRequestMsg">
2180   <part name="body" element="tns:myMethod"/>
2181 </message>
2182 <message name="myMethodResponseMsg">
2183   <part name="body" element="tns:myMethodResponse"/>
2184 </message>
2185
2186 <message name="myOtherMethodRequestMsg">
2187   <part name="body" element="tns:myOtherMethod"/>
2188 </message>
2189 <message name="myOtherMethodResponseMsg">
2190   <part name="body" element="tns:myOtherMethodResponse"/>
2191 </message>
2192
2193 <portType name="MyServicePortType">
2194   <operation name="myMethod">
2195     <input message="tns:myMethodRequestMsg"/>
2196     <output message="tns:myMethodResponseMsg"/>
2197   </operation>
2198   <operation name="myOtherMethod">
2199     <input message="tns:myOtherMethodRequestMsg"/>
2200     <output message="tns:myOtherMethodResponseMsg"/>
2201   </operation>
2202 </portType>
2203
2204 <binding name="MyServiceBinding" type="tns:MyService">
2205   <soap:binding style="document"
2206     transport="http://schemas.xmlsoap.org/soap/http"/>
2207   <operation name="myMethod">
2208     <soap:operation soapAction="MyService#myMethod"/>
2209     <input>
2210       <soap:body use="literal"/>
2211     </input>
2212     <output>
2213       <soap:body use="literal"/>
2214     </output>
2215   </operation>
2216   <operation name="myOtherMethod">
2217     <soap:operation soapAction="MyService#myOtherMethod"/>
2218     <input>
2219       <soap:body use="literal"/>
2220     </input>

```



```

2221     <output>
2222         <soap:body use="literal"/>
2223     </output>
2224 </operation>
2225 </binding>
2226
2227 <service name="MyService">
2228     <port name="MyServicePort" binding="tns:MyServiceBinding">
2229         <soap:address location="http://server:9090/MyService"/>
2230     </port>
2231 </service>
2232 </definitions>
2233

```

2234 If multiple classes are defined in the single C++ header file, the class to be mapped must be  
 2235 specified by name.

2236 This specification requires support for generating a SOAP 1.1, document/literal style binding.  
 2237 Support for additional bindings (such as SOAP 1.2) is not required, however if provided should be  
 2238 consistent with the SOAP 1.1 binding specified in this document. Support for additional binding  
 2239 styles is implementation dependent.

2240

## 2241 **6.5. Method mapping**

### 2242 **6.5.1. Default parameter value mapping**

2243 Where default values are defined in the parameters of a method, these are reflected in the  
 2244 schema as non-required elements. Default values in C++ method prototypes are generally  
 2245 provided to allow users to ignore the parameters.

2246 E.g. a method prototype:

2247

```
2248 long myMethod(char* name, int id = 0, double value = 12.34);
```

2249

2250 would generate a schema like:

2251

```

2252 <xsd:element name="myMethod">
2253     <xsd:complexType>
2254         <xsd:sequence>
2255             <xsd:element name="name" type="xsd:string"/>
2256             <xsd:element name="id" type="xsd:int" minOccurs="0"/>
2257             <xsd:element name="value" type="xsd:double" minOccurs="0"/>
2258         </xsd:sequence>
2259     </xsd:complexType>
2260 </xsd:element>

```

### 6.5.2. Non-named parameters and the return type

Above, we have seen method prototypes with named parameters. C++ allows prototype parameters to be unnamed, simply typed (e.g. `long myMethod(char*, int, double)`). Prototypes defined in this way are not supported.

The return type in C++ methods is unnamed, so, as has been shown above, a name must be generated for the elements required by doc-lit-wrapped WSDL. E.g. for the method prototype above, the response data will be returned using the following schema:

```

2268
2269 <xsd:element name="myMethodResponse">
2270   <xsd:complexType>
2271     <xsd:sequence>
2272       <xsd:element name="myMethodResponseData" type="xsd:int"/>
2273     </xsd:sequence>
2274   </xsd:complexType>
2275 </xsd:element>

```

### 6.5.3. The void return type

Handling of the void return type is controlled by the `oneWay` annotation. If `oneWay` is true, the operation will be mapped to a one-way (in-only) WSDL operation, otherwise it will be mapped to a request-response WSDL operation where the output message is empty.

```

2280
2281 void myMethod(char* name, double value);
2282

```

would generate a schema like:

```

2283
2284
2285 <xsd:element name="myMethodRequestMsg">
2286   <xsd:complexType>
2287     <xsd:sequence>
2288       <xsd:element name="name" type="xsd:string"/>
2289       <xsd:element name="value" type="xsd:double"/>
2290     </xsd:sequence>
2291   </xsd:complexType>
2292 </xsd:element>
2293
2294 <xsd:element name="myMethodResponseMsg">
2295   <xsd:complexType/>
2296 </xsd:element>
2297

```

and a WSDL operation in the WSDL `portType` and binding elements such as:

```

2298
2299
2300 <portType name="MyServicePortType">
2301   <operation name="myMethod">

```

```

2302     <input message="tns:myMethodRequestMsg"/>
2303     <output message="tns:myMethodResponseMsg"/>
2304 </operation>
2305 </portType>
2306
2307 <binding name="MyServiceBinding" type="tns:MyService">
2308     <soap:binding style="document"
2309         transport="http://schemas.xmlsoap.org/soap/http"/>
2310     <operation name="myMethod">
2311         <soap:operation soapAction="MyService#myMethod"/>
2312         <input>
2313             <soap:body use="literal"/>
2314         </input>
2315         <output>
2316             <soap:body use="literal"/>
2317         </output>
2318     </operation>
2319 </binding>

```

2320

2321 Alternatively, if the oneWay annotation is specified on the method:

2322

```

2323 // @oneWay
2324 void myMethod(char* name, double value);

```

2325

2326 the following schema would be generated:

2327

```

2328 <xsd:element name="myMethodRequestMsg">
2329     <xsd:complexType>
2330         <xsd:sequence>
2331             <xsd:element name="name" type="xsd:string"/>
2332             <xsd:element name="value" type="xsd:double"/>
2333         </xsd:sequence>
2334     </xsd:complexType>
2335 </xsd:element>

```

2336

2337 and a WSDL operation in the WSDL portType and binding elements that contains no output  
 2338 element, such as:

2339

```

2340 <portType name="MyServicePortType">
2341     <operation name="myMethod">
2342         <input message="tns:myMethodRequestMsg"/>

```

```

2343     </operation>
2344 </portType>
2345
2346 <binding name="MyServiceBinding" type="tns:MyService">
2347     <soap:binding style="document"
2348         transport="http://schemas.xmlsoap.org/soap/http"/>
2349     <operation name="myMethod">
2350         <soap:operation soapAction="MyService#myMethod"/>
2351         <input>
2352             <soap:body use="literal"/>
2353         </input>
2354     </operation>
2355 </binding>
2356

```

#### 2357 6.5.4. No Parameters Specified

2358 If a C++ method prototype has no parameters, the input schema element is still required (for  
 2359 doc-lit-wrapped WSDL) but is empty. E.g. a method prototype:

```

2360
2361 int getValue();
2362

```

2363 would generate a schema like:

```

2364
2365 <xsd:element name="getValue">
2366     <xsd:complexType/>
2367 </xsd:element>
2368
2369 <xsd:element name="getValueResponse">
2370     <xsd:complexType>
2371         <xsd:sequence>
2372             <xsd:element name="getValueResponseData" type="xsd:int"/>
2373         </xsd:sequence>
2374     </xsd:complexType>
2375 </xsd:element>

```

#### 2376 6.5.5. In/Out Parameters

2377 In/Out parameters allow the method to receive and change the value of a parameter with those  
 2378 changes being subsequently available in the invoking code. In/Out parameter are not needed for  
 2379 remotable calls so are not supported in this mapping.

#### 2380 6.5.6. Public Methods

2381 All public methods of a C++ header will be converted to WSDL operation definitions.

2382 **6.5.7. Inherited Public Methods**

2383 Public methods inherited by a C++ class will not be converted to WSDL operation definitions. If  
2384 an inherited method is required, it must be re-specified in the inheriting class.

2385 **6.5.8. Protected/Private Methods**

2386 Protected and private methods will not be converted to WSDL operation definitions.

2387 **6.5.9. Constructors/Destructors**

2388 Constructors and destructors will not be converted to WSDL operation definitions. The lack of  
2389 state in standard web services makes explicit construction/destruction operations meaningless.

2390 **6.5.10. Overloaded Methods**

2391 Overloaded methods are not supported due to the lack of support for overloading in WSDL 1.

2392 **6.5.11. Operator overloading**

2393 Overloaded operators ( "==" , ">=" , "new" , etc) are not supported.

2394 **6.5.12. Exceptions**

2395 C++ method prototypes can specify that particular exceptions may be thrown by the method.  
2396 Handling of C++ exception throw specifications is not defined by this mapping, and is  
2397 implementation dependent.

---

## 7. References

---

2398

2399

2400

[1] SDO 2.1 Specification

2401

<http://www.osoa.org/display/Main/Service+Data+Objects+Specifications>

2402

2403

[2] SCA Assembly Specification

2404

<http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications>

2405

2406

[3] OMG WSDL to C++ Mapping Specification

2407

<http://www.omg.org/docs/ptc/06-08-01.pdf>

2408