

SCA Service Component Architecture

Java Component Implementation Specification

SCA Version 1.00, 15 February 2007

Technical Contacts:

Ron Barack	SAP AG
Michael Beisiegel	IBM Corporation
Henning Blohm	SAP AG
Dave Booz	IBM Corporation
Jeremy Boynes	Independent
Ching-Yun Chao	IBM Corporation
Adrian Colyer	Interface21
Mike Edwards	IBM Corporation
Hal Hildebrand	Oracle
Sabin Ielceanu	TIBCO Software, Inc
Anish Karmarkar	Oracle
Daniel Kulp	IONA Technologies plc.
Ashok Malhotra	Oracle
Jim Marino	BEA Systems, Inc.
Michael Rowley	BEA Systems, Inc.
Ken Tam	BEA Systems, Inc
Scott Vorthmann	TIBCO Software, Inc
Lance Waterman	Sybase, Inc.

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, 2006, 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 and display 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/download/attachments/35/SCA_AssemblyModel_V100.pdf

2. The full text of this 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, 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

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	ii
1. Java Implementation Type.....	1
1.1. Introduction	1
1.2. Java Implementation Type	1
1.2.1. Services	1
1.2.2. References	4
1.2.3. Dynamic Reference Access	5
1.2.4. Properties.....	5
1.2.5. Implementation Instance Instantiation.....	5
1.2.6. Implementation Scopes and Lifecycle Callbacks.....	7
1.2.7. Specifying the Java Implementation Type in an Assembly.....	8
Specifying the Component Type	9
2. Appendix.....	10
2.1. References.....	10

1. Java Implementation Type

1.1. Introduction

This specification extends [the SCA Assembly Model \[1\]](#) by defining how a Java class provides an implementation of an SCA component and how that class is used in SCA as a component implementation type.

This specification requires all the annotations and APIs as defined by the [SCA Java Common Annotations and APIs specification \[2\]](#). All annotations and APIs referenced in this document are defined in the former unless otherwise specified. Moreover, the semantics defined in the Common Annotations and APIs specification are normative.

1.2. Java Implementation Type

This section specifies how a Java class provides an implementation of an SCA component, including its various attributes such as services, references, and properties. In addition, it details the use of metadata and the Java API defined [in \[2\]](#) in the context of a Java class used as a component implementation type,

1.2.1. Services

A component implementation based on a Java class may provide one or more services.

The services provided by a Java-based implementation may have an interface defined in one of the following ways:

- A Java interface
- A Java class
- A Java interface generated from a Web Services Description Language [3] (WSDL) portType.

Java implementation classes must implement all the operations defined by the service interface. If the service interface is defined by a Java interface, the Java-based component can either implement that Java interface, or implement all the operations of the interface.

A service whose interface is defined by a Java class (as opposed to a Java interface) is not remotable. Java interfaces generated from WSDL portTypes are remotable, see the [WSDL 2 Java and Java 2 WSDL](#) section of the SCA Java Common Annotations and API Specification for details.

A Java implementation type may specify the services it provides explicitly through the use of `@Service`. In certain cases as defined below, the use of `@Service` is not required and the services a Java implementation type offers may be inferred from the implementation class itself.

1.2.1.1. Use of `@Service`

Service interfaces may be specified as a Java interface. A Java class, which is a component implementation, may offer a service by implementing a Java interface specifying the service contract. As a Java class may implement multiple interfaces, some of which may not define SCA services, the `@Service` annotation can be used to indicate the services provided by the implementation and their corresponding Java interface definitions.

The following is an example of a Java service interface and a Java implementation, which provides a service using that interface:

46 Interface:

```
47     public interface HelloService {
48
49         String hello(String message);
50     }
51
```

52 Implementation class:

```
53     @Service(HelloService.class)
54     public class HelloServiceImpl implements HelloService {
55
56         public String hello(String message) {
57             ...
58         }
59     }
60
```

61 The XML representation of the component type for this implementation is shown below for illustrative
62 purposes. There is no need to author the component type as it can be reflected from the Java class.

```
63
64     <?xml version="1.0" encoding="ASCII"?>
65     <componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">
66
67         <service name="HelloService">
68             <interface.java interface="services.hello.HelloService"/>
69         </service>
70
71     </componentType>
72
```

73 The Java implementation class itself, as opposed to an interface, may also define a service offered by a
74 component. In this case, @Service may be used to explicitly declare the implementation class defines the
75 service offered by the implementation. In this case, a component will only offer services declared by
76 @Service. The following illustrates this:

```
77
78     @Service(HelloServiceImpl.class)
79     public class HelloServiceImpl implements AnotherInterface {
80
81         public String hello(String message) {
82             ...
83         }
84
85         ...
86     }
87
```

87 In the above example, HelloWorldServiceImpl offers one service as defined by the public methods on the implementation
88 class. The interface AnotherInterface in this case does not specify a service offered by the component. The following is
89 an XML representation of the introspected component type:

```
90     <?xml version="1.0" encoding="ASCII"?>
91     <componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">
92
93         <service name="HelloService">
94             <interface.java interface="services.hello.HelloServiceImpl"/>
95         </service>
96
97     </componentType>
98
```

99 @Service may be used to specify multiple services offered by an implementation as in:

```
100
101 @Service(interfaces={HelloService.class, AnotherInterface.class})
102 public class HelloServiceImpl implements HelloService, AnotherInterface {
103
104     public String hello(String message) {
105         ...
106     }
107     ...
108 }
```

109

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

```
111 <?xml version="1.0" encoding="ASCII"?>
112 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
113
114     <service name="HelloService">
115         <interface.java interface="services.hello.HelloService"/>
116     </service>
117     <service name="AnotherService">
118         <interface.java interface="services.hello.AnotherService"/>
119     </service>
120
121 </componentType>
```

122 **1.2.1.2. Local and Remotable services**

123 A Java service contract defined by an interface or implementation class may use @Remotable to declare
124 that the service follows the semantics of remotable services as defined by the SCA Assembly Specification.
125 The following example demonstrates the use of @Remotable:

```
126 package services.hello;
127
128 @Remotable
129 public interface HelloService {
130
131     String hello(String message);
132 }
133
```

134 Unless @Remotable is declared, a service defined by a Java interface or implementation class is inferred to
135 be a local service as defined by the SCA Assembly Model Specification.

136

137 If an implementation class has implemented interfaces that are not decorated with an @Remotable
138 annotation, the class is considered to implement a single **local** service whose type is defined by the class
139 (note that local services may be typed using either Java interfaces or classes).

140 An implementation class may provide hints to the SCA runtime about whether it can achieve pass-by-value
141 semantics without making a copy by using the @AllowsPassByReference..

142

143 **1.2.1.3. Introspecting services offered by a Java implementation**

144 In the cases described below, the services offered by a Java implementation class may be determined
145 through introspection, eliding the need to specify them using @Service. The following algorithm is used to
146 determine how services are introspected from an implementation class:

147 *If the interfaces of the SCA services are not specified with the @Service annotation on the implementation
148 class, it is assumed that all implemented interfaces that have been annotated as @Remotable are the*

149 *service interfaces provided by the component. If none of the implemented interfaces is remotable, then by*
 150 *default the implementation offers a single service whose type is the implementation class.*

151

152 **1.2.1.4. Non-Blocking Service Operations**

153 Service operations defined by a Java interface or implementation class may use @OneWay to declare that
 154 the SCA runtime must honor non-blocking semantics as defined by the SCA Assembly Specification when a
 155 client invokes the service operation.

156 **1.2.1.5. Non-Conversational and Conversational Services**

157 The Java implementation type supports all of the conversational service annotations as defined by the SCA
 158 Java Common Annotations and API Specification: @Conversational, @EndsConversation, and
 159 @ConversationAttributes.

160 The following semantics hold for service contracts defined by Java interface or implementation class. A service
 161 contract defined by a Java interface or implementation class is inferred to be non-conversational as defined by
 162 the SCA Assembly Specification unless it is decorated with @Conversational. In the latter case, @Conversational
 163 is used to declare that a component implementation offering the service implements conversational semantics
 164 as defined by the SCA Assembly Specification.

165 **1.2.1.6. Callback Services**

166 A callback interface is declared by using the @Callback annotation on the service interface implemented by
 167 a Java class.

168 **1.2.2. References**

169 References may be obtained through injection or through the ComponentContext API as defined in the SCA
 170 Java Common Annotations and API Specification. When possible, the preferred mechanism for accessing
 171 references is through injection.

172 **1.2.2.1. Reference Injection**

173

174 A Java implementation type may explicitly specify its references through the use of @Reference as in the
 175 following example:

176

177

```
178     public class ClientComponentImpl implements Client {
179         private HelloService service;
180
181         @Reference
182         public void setHelloService(HelloService service) {
183             this.service = service;
184         }
185     }
186
```

187 If @Reference marks a public or protected setter method, the SCA runtime is required to provide the
 188 appropriate implementation of the service reference contract as specified by the parameter type of the
 189 method. This must done by invoking the setter method an implementation instance. When injection occurs
 190 is defined by the scope of the implementation. However, it will always occur before the first service method
 191 is called.

192 If @Reference marks a public or protected field, the SCA runtime is required to provide the appropriate
 193 implementation of the service reference contract as specified by the field type. This must done by setting
 194 the field on an implementation instance. When injection occurs is defined by the scope of the
 195 implementation.

196 If @Reference marks a parameter on a constructor, the SCA runtime is required to provide the appropriate
 197 implementation of the service reference contract as specified by the constructor parameter during
 198 instantiation of an implementation instance.

199 References may also be determined by introspecting the implementation class according to the rules
 200 defined in Section **Error! Reference source not found..**

201 References may be declared optional as defined by the Java Common Annotations and API Specification.

202 **1.2.2.2. Dynamic Reference Access**

203 References may be accessed dynamically through `ComponentContext.getService()` and
204 `ComponentContext.getServiceReference(..)` methods as described in the Java Common Annotations and
205 API Specification.

206 **1.2.3. Properties**

207 **1.2.3.1. Property Injection**

208

209 Properties may be obtained through injection or through the `ComponentContext` API as defined in the SCA
210 Java Common Annotations and API Specification. When possible, the preferred mechanism for accessing
211 properties is through injection.

212 A Java implementation type may explicitly specify its properties through the use of `@Property` as in the
213 following example:

214

215

```
216     public class ClientComponentImpl implements Client {
217         private int maxRetries;
218
219         @Property
220         public void setRetries(int maxRetries) {
221             this.maxRetries = maxRetries;
222         }
223     }
224
```

225

226 If `@Property` marks a public or protected setter method, the SCA runtime is required to provide the
227 appropriate property value. This must done by invoking the setter method an implementation instance.
When injection occurs is defined by the scope of the implementation.

228 If `@Property` marks a public or protected field, the SCA runtime is required to provide the appropriate
229 property value. When injection occurs is defined by the scope of the implementation.

230 If `@Property` marks a parameter on a constructor, the SCA runtime is required to provide the appropriate
231 property value during instantiation of an implementation instance.

232 Properties may also be determined by introspecting the implementation class according to the rules defined
233 in Section **Error! Reference source not found.**

234 Properties may be declared optional as defined by the Java Common Annotations and API Specification.

235 **1.2.3.2. Dynamic Property Access**

236 Properties may be accesses dynamically through `ComponentContext.getProperty()` method as described in
237 the Java Common Annotations and API Specification.

238

239

240 **1.2.4. Implementation Instance Instantiation**

241 A Java implementation class must provide a public or protected constructor that can be used by the SCA
242 runtime to instantiate implementation instances. The constructor may contain parameters; in the presence
243 of such parameters, the SCA container will pass the applicable property or reference values when invoking
244 the constructor. Any property or reference values not supplied in this manner will be set into the field or
245 passed to the setter method associated with the property or reference before any service method is
246 invoked.

247 The constructor to use is selected by the container as follows:

- 248 1. A declared constructor annotated with a `@Constructor` annotation.
- 249 2. A declared constructor that unambiguously identifies all property and reference values.

250 3. A no-argument constructor.

251 The @Constructor annotation must only be specified on one constructor; the SCA container must report an
252 error if multiple constructors are annotated with @Constructor.

253

254 The property or reference associated with each parameter of a constructor is identified:

- 255 • by name in the @Constructor annotation (if present)
- 256 • through the presence of a @Property or @Reference annotation on the parameter declaration
- 257 • by uniquely matching the parameter type to the type of a property or reference

258

259 Cyclic references between components may be handled by the container in one of two ways:

260

- 261 • If any reference in the cycle is optional, then the container may inject a null value during
262 construction, followed by injection of a reference to the target before invoking any service.
- 263 • The container may inject a proxy to the target service; invocation of methods on the proxy may
264 result in a ServiceUnavailableException

265 The following are examples of legal Java component constructor declarations:

266

```
267 /** Simple class taking a single property value */
```

```
268 public class Impl1 {
```

```
269     String someProperty;
```

```
270     public Impl1(String propval) {...}
```

```
271 }
```

272

```
273 /** Simple class taking a property and reference in the constructor;
```

```
274  * The values are not injected into the fields.
```

```
275  */
```

```
276 public class Impl2 {
```

```
277     public String someProperty;
```

```
278     public SomeService someReference;
```

```
279     public Impl2(String a, SomeService b) {...}
```

```
280 }
```

281

```
282 /** Class declaring a named property and reference through the constructor */
```

```
283 public class Impl3 {
```

```
284     @Constructor({"someProperty", "someReference"})
```

```
285     public Impl3(String a, SomeService b) {...}
```

```
286 }
```

287

```
288 /** Class declaring a named property and reference through parameters */
```

```
289 public class Impl3b {
```

```

290     public Impl3b(
291         @Property("someProperty") String a,
292         @Reference("someReference") SomeService b
293     ) {...}
294 }
295
296 /** Additional property set through a method */
297 public class Impl4 {
298     public String someProperty;
299     public SomeService someReference;
300     public Impl2(String a, SomeService b) {...}
301     @Property public void setAnotherProperty(int x) {...}
302 }
303

```

304 1.2.5. Implementation Scopes and Lifecycle Callbacks

305 **The Java implementation type supports all of the scopes defined in the Java Common**
306 **Annotations and API Specification: STATELESS, REQUEST, CONVERSATION, and COMPOSITE.**
307 **Implementations specify their scope through the use of the @Scope annotation as in:**

```

308
309     @Scope("COMPOSITE")
310     public class ClientComponentImpl implements Client {
311         // ...
312     }

```

313 **When the @Scope annotation is not specified on an implementation class, its scope is defaulted**
314 **to STATELESS.**

315 **A Java component implementation specifies init and destroy callbacks by using @Init and**
316 **@Destroy respectively. For example:**

```

317
318     public class ClientComponentImpl implements Client {
319
320         @Init
321         public void init() {
322             //...
323         }
324
325         @Destroy
326         public void destroy() {
327             //...
328         }
329     }
330

```

331 **1.2.5.1.** *Java implementation classes that are CONVERSATION scoped may use @ConversationID to have*
332 *the current conversation ID injected on a public or protected field or setter method. Alternatively,*
333 *the Conversation API as defined in the Java Common Annotations and API Specification may be*
334 *used to obtain the current conversation ID.* **Conversational Implementation**

335 For the provider of a conversational service, there is the need to maintain state data between successive
336 method invocations within a single conversation. For an Java implementation type, there are two possible
337 strategies which may be used to handle this state data:

- 338 1. The implementation can be built as a stateless piece of code (essentially, the code expects a new
 339 instance of the code to be used for each method invocation). The code must then be responsible
 340 for accessing the conversationID of the conversation, which is maintained by the SCA runtime code.
 341 The implementation is then responsible for persisting any necessary state data during the
 342 processing of a method and for accessing the persisted state data when required, all using the
 343 conversationID as a key.
- 344 2. The implementation can be built as a stateful piece of code, which means that it stores any state
 345 data within the instance fields of the Java class. The implementation must then be declared as
 346 being of [conversation scope](#) using the @Scope annotation. This indicates to the SCA runtime that
 347 the implementation is stateful and that the runtime must perform correlation between client method
 348 invocations and a particular instance of the service implementation and that the runtime is also
 349 responsible for persisting and restoring the implementation instance if the runtime needs to clear
 350 the instance out of memory for any reason. (Note that conversations are potentially very long lived
 351 and that SCA runtimes may involve the use of clustered systems where a given instance object may
 352 be moved between nodes in the cluster over time, for load balancing purposes)
 353

354 1.2.6. Accessing a Callback Service

355 **Java implementation classes that require a callback service may use @Callback to have a**
 356 **reference to the callback service associated with the current invocation injected on a public or**
 357 **protected field or setter method.**

358 1.2.7. Semantics of an Unannotated Implementation

359 The section defines the rules for determining properties and references for a Java component
 360 implementation that does not explicitly declare them using @Reference or @Property.

361 In the absence of @Property and @Reference annotations, the properties and references of a class are
 362 defined according to the following rules:

- 363 1. Public setter methods that are not included in any interface specified by an @Service annotation.
 364 2. Protected setter methods
 365 3. Public or protected fields unless there is a public or protected setter method for the same name
 366

367 The following rules are used to determine whether an unannotated field or setter method is a property or
 368 reference:

- 369 1. If its type is simple, then it is a property.
 370 2. If its type is complex, then if the type is an interface marked by @Remotable, then it is a reference;
 371 otherwise, it is a property.
 372 3. Otherwise, if the type associated with the member is an array or a java.util.Collection, the basetype is
 373 the element type of the array or the parameterized type of the Collection; otherwise the basetype is the
 374 member type. If the basetype is an interface with an @Remotable or @Service annotation then the
 375 member is defined as a reference. Otherwise, it is defined as a property.

376 The name of the reference or of the property is derived from the name found on the setter method or on
 377 the field.

378

379 1.2.8. Specifying the Java Implementation Type in an Assembly

380 The following defines the implementation element schema used for the Java implementation type:.

381

```
382 <implementation.java class="NCName" />
```

383

384 The implementation.java element has the following attributes:

- 385 • **class (required)** – the fully qualified name of the Java class of the implementation

386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409

1.2.9. Specifying the Component Type

For a Java implementation class, the component type is typically derived directly from introspection of the Java class .

A component type can optionally be specified in a side file. The component type side file is found with the same classloader that loaded the Java class. The side file must be located in a directory that corresponds to the namespace of the implementation and have the same name as the Java class, but with a .componentType extension instead of the .class extension.

The rules on how a component type side file adds to the component type information reflected from the component implementation are described as part of [the SCA assembly model specification \[1\]](#). If the component type information is in conflict with the implementation, it is an error.

If the component type side file specifies a service interface using a WSDL interface, then the Java class should implement the interface that would be generated by the JAX-WS mapping of the WSDL to a Java interface. See the [section 'WSDL 2 Java and Java 2 WSDL' in \[2\]](#).

410
411
412
413
414
415
416
417
418
419

2. Appendix

2.1. References

[1] SCA Assembly Specification

http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[2] SCA Java Common Annotations and APIs

http://www.osoa.org/download/attachments/35/SCA_JavaCommonAnnotationsAndAPIs_V100.pdf