SiDiff 2.0 – Service-Howto

Sven Wenzel

26. Oktober 2009

Inhaltsverzeichnis

Documentation

This document is maintained continuously, the subsequent table provides information about changes.

Datum	Änderungen
16.2.09	erste Version (einfache und instanzierbare Services)
17.2.09	Kochrezept für konfigurierbare Services Hinweis auf Bundle-Namen als String-Konstanten Kapitel über häufige Stolperfallen
18.2.09	Kochrezept für konfigurierbare, instanzierbare Services
23.2.09	Neues Kapitel: Erstellen einer Laufzeitumgebung Neues Kapitel: Zusammenspiel von Services
17.6.09	Überarbeitet: Zusammenspiel von Services: der Service-Kontext Kochrezepte B und C an aktuellen ServiceHelper angepasst
17.09.09	Komplett überarbeitet
22.09.09	Korrektur kleinerer Fehler **Dokument ist auf Stand der Implementierung (rev. 1470)**
26.10.09	englische Übersetzung eingepflegt

1 Introduction

1.1 Why services?

SiDiff is not only an algorithm to compute the differences but rather a complete toolbox which allows one to build different tools in the field of model differencing. Moreover, it is not a product but rather a subject of ongoing research. SiDiff has been realized on the basis of services in order to achive a flexibility in particular in order to allow a single components to be exchanged.

1.2 Basic concept

The basic concept is that each group of functions offered by SiDiff which might eventually be replaced by another implementaion is realized as a seperate service. Thus, there are services for:

- The reading of documents
- The annotaitons of the model elements with artifial attributes for example: path or matrix values
- Computation of similarities
- Matching
- ... and so on.

In addition services can be seperated over several modules. For examplem the annotation is further subdivided into:

- Extension of the data model in order to be able to append the annotations to model elements,
- the algorithm which scans all model elements and annotates them,
- and single annotation functions which compute artificial attribute values.

1.3 Technical realization

Technically, services and modules are realized as OSGI bundles. We are using Eclipse Equinox as OSGI platform.

A *Bundle* is a collection of Java classes and further resources.

! In Eclipse OSGi-Bundles and Pulgins are treated equally. In order to create a new bundle one should select the Eclipse wizard "New Plug-In-Project". It is important that under the item *Target Platform* in OSGi-Framework **standard** is chosen. Both variants defer in the dependency to the OSGi-Framework. If one chooses standard, only the package `org.osgi.framework` is required. It is completely left open which bundle provides this package (in other words which OSGi-Implemetation is used), if one choses the variant "Equinox", one is independent from the bundle `org.eclipse.osgi` (and in addition from the OSGi-Implemetation Equinox).

A *Service* is a function which is provided somehow. We distinguish between the service definition (i.e. Interface) and the realization. The interface is essentially only a java interface and the realization implements this interface.

There are at least two bundles for each service. One bundle defines the service (i.e. the interface) and may offer classes and resources which are used by all realizations. At least one bundle provides an implementation of the service. Of cource the same service can have different implementaions.

1.4 Kinds of Services

We distinguish the following types of services:

- Simple services (see recipe A)
- Services which can be instantiated (see recipe B)
- Services which can be configured (see recipe C)
- Services which can be configured and instantiated (see recipe D)

A *simple Service* is a service which is provided by *one Object* which defines one particular interface.

A *Service which can be instantiated* is a service which is provided by a new object whenever the service is called. This concept is similar to the OSGi-concept of a `ServiceFactory`, however it defers substantially because *at each and every* invocation of the service a new service object is created. Even then if the service is invoked several times from the same bundle. In order to distinguish it from a `ServiceFactory` we use a `ServiceProvider` here.[1].

Services which can be configured are services which can be adapted to a specific document type using suitable data (for example a configuration file). In this case a configurator is registered and using this configurator a service instance can be created

[1]From the point of view of OSGi, a `ServiceProvider` is a service which creates only one new object and returns it. We call this object an implementaion of the service because the mechanism of the `ServiceProvider` works transparently

and configured. Afterwards this instance is configured permanently and can be used repeatedly. In this way several instances of a service which are configured differently can coexist at the same time.

Services which can be configured and instantiated are a mixture of the both previous service types. The services are configured and a new object is created at each invocation of the service. Essentailly the `ServiceProvider` is configured here.

1.5 Hierarchies of services

Services can be classified hierarichally. A first set of service classes has been introduced in the previous section 1.4. It is easy to see that services can be structured hierarchically because each service which can be configured is also simple service with additional properties. Another example from the SiDiff is the matching service which has two specific variants in the form of the hash matching service and the iterative matching service.

The service hierarchy is is taken into account in the context of the service-management as follows. Each service is not only registered under its own interface but also under the interfaces of its superclasses. For example, the hash matching service is not only registered as HashMatchingService but also as MatchingServicen in general. This means also that a client which requests a MatchingService gets a service returned. Normally the least specific service is delivered by the ServiceHelper. If two services are equally specific, in other words they are located at the same level at the service hierarchy, then one of them is returend).

1.6 The ServiceHelper

The ServiceHelper encapsulates the service layer of the OSGi-Framework. We are using the ServiceHelper as a general interface for registering and requesting services. One should not directly invoke operations of the OSGi-Framework using the OSGi-BundleContext. If the ServiceHelper is used the system can be better maintained and all services are used consistently.

1.7 Literature

We recommend as an introduction to the OSGi topic the book:
Gerd Wütherich et.al.,
Die OSGi Service Platform,
dpunkt Verlag,
ISBN 978-3-89864-457-0,
2008

2 Conventions

2.1 Seperation of Interface and Implementaion

We have already mentioned above that there are at least two bundles for each service in order to sepearte interface and implementation.

There can be exceptions to this rule in which the interface and the implementaion of a service are not seperated. Such exceptions occur in particular if there can be at most one implementaion of a service and if it can not be exchanged. However, such exceptions should be discussed together.

2.2 Choice of Names

Each service should define its name space and the bundle has to be named accordingly. The new name starst always with `org.sidiff.`. Subsequently follows `core.` if the bundle belongs to the SiDiff kernel. Then follows the bundle name and such.

Example: Matching (which belongs to the kernel) has the name: → `org.sidiff.core.matching`, Fingerprints (which is not part of the kernel) has the name: → `org.sidiff.fingerprints`.

If a bundle has several subbundles or if a service has several implementaions one has to append the names of these packages.

Example: `org.sidiff.core.matching.idbased` and `org.sidiff.core.matching.iterative`

2.3 Visibilities

OSGi allows one to define in the manifest file of a bundle which packages are visible from outside and which can be used internally within the bundle. This mechanism should be used as much as possible. As a result only those classes should be visible from the outside which are really used. In order to hide classes one can, if necessary, create a subpackage `internal` which is not accessible from the outside.

3 Cooking recipe A: Offering and Using a Simple Service

In order to offer a service two bundles should be created. One of them with the Service-Interface and the other one with the implemetation of the service.

3.1 Creation of the Interface-Bundle

The Interface-Bundle is created as a bundle which provides classes and resources (see white paper "Introduction to the SiDiff 2.0 Development Environment").

Essentially this bundle provides the Java-Interface. In addition one can offer further classes and resources here. For example: Auxilary functions which can be used by different implementations.

Definition of the Service-Interface The Service-Interface is a pure Java interface which extends the interface `org.sidiff.common.services.Service`. All methods which service offers should be defined here.

```
public interface MyService extends Service {
  public void doSomething();
  public boolean doAnotherThing(int i);
}
```

! Abstract classes should not be used as service interfaces. If one really wants to provide abstract behaviour one should create an abstract class in addition to the interface in order to be able to inherit from concrete service implementations

3.2 Creation of the Implementation Bundle

The bundle with the implementaiton should be created as described in the white paper "Introduction to the SiDiff 2.0 Development Environment", however with one important difference.

In step 3 , one should choose to generate an activator. The name which is proposed should be kept.

Dependencies from the Interface-Bundle Before implementing a service, one has to specify the Interface-Bundle as *Required Plugin* in the *Dependencies* section of the manifest. In this way classes and interfaces of the Interface-Bundle can be used.

Implementing with the Service In order to implement the service a new class has to be created which implements the interface from the Interface-Bundle. Of course one can arbitrarily create and use many other classes. One can also use other services.

```
public class MyServiceImpl implements MyService {
  public void doSomething() {
    System.out.println("something");
  }
  public boolean doAnotherThing(int i) {
    return (i<5);
  }
}
```

Publishing a Service-Implementation using the Activator A concrete service can only be used if it has been registered before. This is done by the central OSGi-Management which is accessible via the BundleContext. The BundleContext is obtained as a parameter in the start method of the activator. The service implementation is also registered here.

! Only implementations of services are registered. Service-Interfaces are assumed to be known whenever necessary and are not explicitly registered

It follows an example activator. `MyService.class` contains a Java interface which is defined in an Interface-Bundle.

```
public class Activator implements BundleActivator {

  public static final String BUNDLE_NAME = "org.sidiff.example.myservice";

  @Override
  public void start(BundleContext context) throws Exception {
    ServiceHelper.registerService(context, MyService.class,
      new MyServiceImpl(), null, null);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
  }

}
```

The `start()`-Method is invoked automatically by the OSGi-Framework if the bundle is started.

A service is registered using `registerService()`. This operation has the following parameters:

1. The OSGI-BundleContext
2. The service whose implementation is offered here

3. One **Instance** of the Service-Implemetation.
4. The DocumetnType which is supported by this particular service or a `null`, if this service is independent from documet types.
5. An optional name of variant or `null`.
6. As another optional parameter one can supply a diciotnary of key-value pairs as for all OSGi-Services. This dicitionary starts in the OSGi-ServiceRegistry.

If necessary the `stop()`-Method can contain instructions which are to be executed if the bundle is removed from the system.

> ! All bundles should store their name which is at the same time Projectname as a string constant in order to be able to refrence it. This constant should, if possible, be defined in the activator in order to enable one to find it quickly within the project.

3.3 Using the Service

If a service has been implemented as described above it can be used by other bundles. The bundle with the service interface must be specified in the manifest of the using bundle. It is not necessary to register the implementation bunlde and in fact registering implemetation bundles should be avoided completely in order to make implementaitons exchangable.[2]

In additon the bundle `org.sidiff.common.services` must be registered as *Required Plugin*. It provides auxilary functions such as `ServiceHelper`.

The implementation of a service can be invoked in the using bundle as follows:

```
MyService ms = ServiceHelper.getService(context, MyService.class,
 null, null);
if (ms != null) {
  ms.doSomething();
}
```

`MyService` represents the Java-Interface which defines the Service-Interface. The concrete implementation is not known here. The parameter `context` is the Bundle-Context of OSGi. The second parameter is the Interface of the requested service. The third parameter is optional, it can specify the document type for which the service is required. The forth parameter can choose one of the variant. In simple cases the third and forth parameters are `null`.

[2] Which implementation is availabe can be seen from the current run-time environment. For example in an Eclipse-Runtime-Environment for debugging this specified by the selection of the plugins and bundles which are to be loaded.

! On the procedures presented here one object has been registered as implementation of the service. Thus all users of the service access the same object. This must be considered in the definition of the interface and the implemetation (e.g. with respect to temporary data).

4 Cooking Recipe B: Definition and Use of a Providable Service

Sometimes a service must be used concurently in differnt contexts. In such cases it is not suitable to register exactly one instance (i.e one object) in the OSGi-Framework. `ProvideableService` and `ServiceProvider` are concepts for creating service instances only at the time when the service is resquested. This concept also allows each user to have an instance of his own. Both classes are supplied in the bundle `org.sidiff.common.services`.

> ! OSGi provides a so called `ServiceFactory`. This is an interface which can be implemented by a service. In this case if a service is requested by different bundles, the factory is used and for each bundle a new object is created. The service instances which are created using the `ServiceFactory` are transparently stored by the OSGi-Framework for each bundle. Hence, two different instances of the service are provided with the same service instance. Concurrent execution and seperation of information is thereforenot possible. This is why we don't use the concept of a `ServiceFactory`.

4.1 Definition of a Service-Interface

The Service-Interface is defined in the same way as with simple services. However the interface must in addition inherit from the interface `ProvideableService`. This interface is actually empty and serves only as a type of marker in the automatic process of instantiation.

```
public interface MyService extends ProvideableService {
  public void doSomething();
  public boolean doAnotherThing(int i);
}
```

In addition the definition of a `ServiceProvider` is requested which defines the method `createInstance()`.

```
public interface MyServiceProvider extends ServiceProvider<MyService> {
}
```

It is important that the interface follows the following conventions:

1. The name is the same as the one of the service interface with the additonal suffix `Provider`.
2. It is located in the same package like the Servie-Interface.
3. It inherits from the interface `ServiceProvider`. and is typecast to the Service-Interface.

4.2 Implementaion of a Service

The service is implemented as usual by implementing the interface.

In addition the implementation of service provider is necessary. It has a method `createInstance` which is called in order to create an instance of the service.

```
public class MyServiceProviderImpl implements MyServiceProvider {
  @Override
  public MyService createInstance() {
    return new MyServiceImpl();
  }
}
```

This method must return an object whose type is this service.

4.3 Publishing the Service-Implementation

A Service-Implementation is published via the activator. This time however the `ServiceProvider` is registerd. All other parameters are the same as indicates when a simple service is registerd.

```
public class Activator implements BundleActivator {

  @Override
  public void start(BundleContext context) throws Exception {
    ServiceHelper.registerServiceProvider(context,
      MyServiceProvider.class, new MyServiceProviderImpl(),
      null, null);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
  }

}
```

It is not necessary to register the service implementaion itself. It is provided by the `ServiceProvider`.

4.4 Usage of the Service

A service which is implemented as described above can be used in other bundles in the same way like simple services. As preparation one must specify in the manifest a dependency from the Interface-Bundle. Moreover the dependency from the bundle `org.sidiff.common.services` must be specified.

The `ServiceProvider` is accessed transparently via the `ServiceHelper`. The using bundle can request the implementation of the service in the same way as a simple service.

```
MyService ms = ServiceHelper.getService(context, MyService.class, null ,null);
if (ms != null) {
  ms.doSomething();
}
```

`MyService` represents again the Java-Interface which defines the Service-Interface. Nothing is known about the provider being used. The provider is transparently accessed, and also the service implementation is instantiated transparently.

The `ServiceProvider` checks whether the required service is a `ProvideableService`. If so, the respectively named `ServiceProvider` (in this case `MyServiceProvider`) is searched in the OSGi-Framework and it's `createInstance()` method is called. The details of this procedure is are not visible to user of the service due to the encapsulation in the `ServiceHelper`.

! With the techniques presented here, always a new object is returened as service implementation. If this particular object is required several times one should create a refrence to this object after the intial service request.

5 Cooking Recipe C: Services which can be Configured

In many cases one needs services which can be configured for one particular documet type. In addition we want to have several services of the same type which are configured differently to exist in parallel.

Several service instances which are differently configured can exist in parallel. The instances are distinguished by the document type to which they belong. SiDiff has been designed in such a way that document of different type can be processed and each document type has its own configuration. This is why we use the document type as key attribute.

In order to support several configurations for the same documet type a variant can be specified in addition to the document type. The variant specified using a simple string, different configurations can be distinguished in this way.

Services need to configured only once and can then be used arbitrary often. The integration in other algorithms is made more efficient this way because initializations must be performed only once.

5.1 Definition of Service which can be Configured

Service-Interface In order to define a configurable service the service interface must extend the interface `ConfigurableService`. It is located in the bundle `org.sidiff.common.services` as other parts which are relavent to services.

```
public interface ConfigurableService {
  public String configure(Object... configData);
  public void deconfigure();
  public Dictionary<String, String> getProperties();
}
```

The method `configure()` is being used in order to configure the service with arbitrary data (`Object...`). This data can be a freely defined set of objects for example of type String or a configuration file. It is shown below how these data are passed. The method `configure()` of a configurable service is called by the ServiceHelper in a way which is transparent for the user of the service.

The `configure()` method returns a document type which is suitable for this configuration. Normally this type is derived from the configuration data, it can only be determined here. This returned type is used as a key to register the service later in the OSGi-Framework.

The method `deconfigure()` can be used to deinitialize configurations. It is called if a service which has already be configured shall be deregisterd.

The operation `getProperties()` returns a String-to-String dictionary which contains further service properties as key value pairs. When a service is registered, the dictionary is carried after the excecution of `configure()` and passed on to the OSGi-Framework. Normally, no further property is defined and the value null can be returend.

Since a configured service is instatiated automatically, it's constructor must not have a parameter.

Implementation This service is implemented in the same way like simple services. The implementation differs only by the additional methods of the `ConfigurableService` which provide the configuration to the service instance.

The following example assumes that we have already produced an interface with the name `MyConfigurableService`. It should have been derived from `ConfigurableService` as described above and should not define addtional methods.

```
public class MyConfigurableServiceImpl implements MyConfigurableService

  String value;

  @Override
  public String configure(Object... configData) {
    value = (String)configData[0];
    return value;
}

  @Override
  public void deconfigure() {
  }

  @Override
  public Dictionary<String, String> getProperties() {
    return null;
  }
}
```

5.2 Publishing a Configurable Service

A configurable service is published by registering the interface and the concrete implementation at the ServiceHelper. This is typically done in the activator:

```
public class Activator implements BundleActivator {

  public void start(BundleContext context) throws Exception {
    ServiceHelper.registerServiceConfigurator(
      context, MyConfigurableService.class, MyConfigurableServiceImpl.class);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
  }

}
```

5.3 Using a Configurable Service

A configurable service is used in two setps. In the first step the service is configured, this needs to be done only once. In the second step the configured instance of the service is used, the second step can be called serveral times.

Configuring a Service A service is configured using the `ServiceHelper` by the following method:

```
public static void configureInstance(
  BundleContext context,
  Class<?> interfaceClass,
  String docType,
  String variant,
  Object... configData) {
```

This code can be called for example as follows:

```
ServiceHelper.configureInstance(
  context
  MyConfigurableService.class,
  eObj.eClass().getEPackage().getNsURI(),
  "SIMPLE",
    "config.xml");
```

The following parameters are passed: the BundleContext, the service interface, the document type, the name of the variant if applicable, and an arbitrary number of further objects as configuration data. It is recommended to pass one configuration file.

After this invocation the service is avilable in a configured form. The configuration as such is excecuted transparently by the `ServiceHelper`.

Requesting a Service In order to use a service which has already been configured it must be requested via the `ServiceHelper` as all other services. However, the documet type and the variant if applicable must be passed as additional parameters.

```
MyService ms = ServiceHelper.getService(context, MyConfigurableService.class,
                                        eObj.eClass().getEPackage().getNsURI());
```

or

```
MyService ms = ServiceHelper.getService(context, MyConfigurableService.class,
                                        eObj.eClass().getEPackage().getNsURI(),
                                        "SIMPLE");
```

! Within the SiDiff-Project we use EMF-Models, therefore it is recommended to use the namespace URI of a metamodel of a document type.

6 Cooking Recipe D: Service which can be Configured and Instantiated

Services which can be instantiated can be additionally be made configurable by additional configuration data. In order to achive this, a `ServiceProvider` which is at the same time a `ConfigurableService` is created. In order to facilitate the use of this concept special Java interfaces are provided which unite both mechanisms.

The `ServiceProvider` can be configured using different sorts of data. Several differently configured providers can coexist. If a concrete service is requested, a new instance is created each time and passed back. It is not necessay to re-read the configuration each time.

6.1 Definition of a Service which can be Configured and Instantiated

Service-Interface The new service must implement the interface `ConfigurableProvideableService` which is only a marker interface again. It serves only for the purpose to transparently delegate the request of a service to the configured `ServiceProvider`.

```
public interface MyService extends ConfigurableProvideableService {
  public void doSomething();
  public boolean doAnotherThing(int i);
}
```

In addtion a `ServiceProvider` is requested. This time, however, a configurable provider is required. It can be created with the interface `ConfigurableProvideableService`.

```
public interface MyServiceProvider
                 extends ConfigurableServiceProvider<MyService> {
}
```

Here again one should follow the naming convention to name the provider like the service with the additonal suffix `Provider`.

Implementation of a Service The service is implemented in the same way like simple services. It is not necessary to take specific behaviour into account. On the contrary unlike with configurable services no additional methods for configuration purposes need to be implemented. At this time the configuration addresses only the `ServiceProvider`.

Implementation of the Provider The `ServiceProvider` has the task to create an new instance of the service at each request and to pass it back. Moreover one should be able to configure it using configuration data. The provider must also implement the interface of a `ConfigurableServiceProvider` which unites `ServiceProvider` and `ConfigurableService`.

```
public class MyServiceProviderImpl implements MyServiceProvider {
  private MyConfiguration config;

  @Override
  public String configure(Object... configData) {
    config = ...
  }

  @Override
  public void deconfigure() {
    // nothing to do
  }

  @Override
  public Dictionary<String, String> getProperties() {
    // no special properties
    return null;
  }

  @Override
  public MyService createInstance() {
    MyService ms = new MyServiceImpl();
    ms.setConfiguration(config);
    return ms;
  }

}
```

The methods `configure()`, `deconfigure()` and `getProperties()` address configurations issues. All information necessary for a configuration should be stored by calling the `configure()` method (indicated in the example by `MyConfiguration`). It does not matter here how the data are managed until the first instatiation of the service. One can also store all supplied objects (`Object... configData`) and to evaluate it at only at that time of the instantition. However, it is important to check that `configure()` returns the document type which is expected in this context.

The `createInstance()` method is called when the service is requested. Here the concrete instance of the service is created, configured if applicable (indicated in the example by `ms.setConfiguration(config)`, alternatively one could use another instance depending on the configuration) and passed back.

The configured `ServiceProvider` is instatiated automatically, therefore it requires a constructor without parameters.

6.2 Publishing the Service

A service which can be configured and instantiated is published by registering the interface and the concrete implementation of the service at the ServiceHelper. This is done typically in the activator:

```
public class Activator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    ServiceHelper.registerServiceConfigurator(
      context, MyServiceProvider.class, MyServiceProviderImpl.class);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
  }

}
```

6.3 Using a Service

A service which can be configured and instatiated is used in the same way like a noraml configurable service.

Configuring the Service The service is configured again using the ServiceHelper according to the following method.

```
public static void configureInstance(
  BundleContext context,
  Class<?> interfaceClass,
  String docType,
  String variant,
  Object... configData) {
```

This is invoked for example as follows

```
ServiceHelper.configureInstance(
  context
  MyConfigurableService.class,
  eObj.eClass().getEPackage().getNsURI(),
  "SIMPLE",
    "config.xml");
```

Using a Service In order to use a completely configured and instantiable service it is requested via the ServiceHelper as all other services.

```
MyService ms = ServiceHelper.getService(context, MyService.class,
                                        eObj.eClass().getEPackage().getNsURI());
```

or

```
MyService ms = ServiceHelper.getService(context, MyService.class,
                                        eObj.eClass().getEPackage().getNsURI(),
                                        "SIMPLE");
```

The call of `getService()` leads to a call of a `createInstance()` method at `ConfigurableServiceProvider`, the latter call creates a concrete instance of the service and passes it back. Thus each call of `getService()` returns a new instance of the configured service. The user does not see any details of how the service is configured and instantiated.

7 Cooperation of Services: Service Context and Communication

Some services in SiDiff are tightly connected to each other. Basically the OSGi-Framework allows us to freely offer services to exchange implementations and to encapsulate details. However, in some cases several service instances belong to each other and can only be used as a group.

In order to support such cases, we have developed the ServiceContext. It allows us to define groups of services which only require each other and offer services for accessing the other services in this context. In addition it supplies mechanisems for the communication between services. It provides an event bus which allows services to send messages and which can be monitored for incomming messages.

The management of correspondences and candidates in SiDiff is a typical example of two services which depend on each other. Whenever a new correspondence is created, the document elements which are involved in this corrrespondence are no longer avalilabe as candidates.

7.1 Context and Context-Sensitive Services

Context are realized by the class `ServiceContext`. Several services can be added to such a class instance.

```
public Object putService(Class<?> serviceId, Object service, int... initParams) {
  ...
}
public boolean containsService(Class<?> serviceID) {
  ...
}
public <X> X getService(Class<X> serviceID) {
  ...
}
public void initialize(Object... params) {
  ...
}
public void setDefaultParams(int... defaultParams) {
  ...
}
```

`putService()` inserts the service instance `service` into a context. This service instance is identified by its service interface `ServiceID`. For each service interface there can exist at most one instance in the context.

`getService()` returns selected services. Before using this service one should check using the operation `containsService()` whether a desired service exists in the pariticular context, because otherwise exceptions may occur.

When all services of a group have been inserted into a context it must be initialized using the operation `initialize()`. The parameter objects which are passed to initialize are passed to all services in this context. A typical example are two models which are used in a comparison. One can specify that particular service obtains only a subset of all parameter objects. This subset is specified by a list of indexes. This list of indexes is specified when a service is inserted into a context using `putService()`. The operation `setDefaultParams()` takes a set of indexes which are used as parameters if `putService()` is invoked without such specific subset.

> **!** As soon as a context has been initialized it must not be changed anymore. If services are inserted or moved later, this will lead to exception.

Context-Sensitive Services Some services must be initialized with certain data which also determine a context. For example, this can be the set of documents which are to be compared. All services in a context must be initialized with the same data. In order to guarantee this the `ServiceContext` triggers all service initialization in the operation `initialize()`.

Services which shall be initialized by the context must implement the interface `ContextSensitiveService`.

```
public interface ContextSensitiveService {
  public void initialize(ServiceContext serviceContext, Object... contextElements);
}
```

The operation `initialize()` passes the context and if applicable further objects for the initialization to this service.

> **!** It is not necessary that all services which belong to a `ServiceContext` implement `ContextSensitiveService`. `ContextSensitiveService` must only be implemented by services which need to access the context.

Examples In the following example the services A and B are inserted into a context. The context is then initialized with two models. Indexes are not used because A and B need the same initalization parameters.

```
ServiceContext serviceContext = new ServiceContext();

serviceA = ServiceHelper.getService(context, ServiceA.class);
serviceContext.putService(ServiceA.class, serviceA);

serviceB = ServiceHelper.getService(context, ServiceB.class);
serviceContext.putService(ServiceB.class, serviceB);

serviceContext.initialize(model1, model2);
```

The example can be further extended by setting Standard-Indices before call the operation `initialize()`. By default all Parameter-Objects are passed to all services by `initialize()`.

```
serviceContext.setDefaultParams(0, 1);
```

Next we assume another service C which needs only the first model and a string as parameter. Now the example look as follows:

```
ServiceContext serviceContext = new ServiceContext();

serviceA = ServiceHelper.getService(context, ServiceA.class);
serviceContext.putService(ServiceA.class, serviceA);

serviceB = ServiceHelper.getService(context, ServiceB.class);
serviceContext.putService(ServiceB.class, serviceB);

serviceC = ServiceHelper.getService(context, ServiceC.class);
serviceContext.putService(ServiceC.class, serviceC, 0, 2);

serviceContext.setDefaultParams(0, 1);
serviceContext.initialize(model1, model2, "abcdefg");
```

Realization of service A which uses the contexts in order to access the other services can look for example as follows:

```
public interface MyService extends ContextSensitiveService {
  ...
}

public class MyServiceImpl implements MyService {

  private ServiceContext context = null;

  private Resource modelA = null;
  private Resource modelB = null;

  @Override
  public void initialize(ServiceContext serviceContext, Object... models) {
    this.modelA = (Resource) models[0];
    this.modelB = (Resource) models[1];
    this.context = serviceContext;
    if (!context.containsService(ServiceB.class)) {
      throw new RuntimeException("benoetigter Service fehlt");
    }
  }

  @Override
  public void doSomething() {
    ServiceB b = this.context.getService(ServiceB.class);
  }

  ...
}
```

7.2 Communication between Services

Services in the same context sometimes must exchange information. `ServiceContext` supports this by several different `EventDispatcher`. Each of them realizes an observer pattern. Each dispatcher manages a set of listeners (observers) for a particular event type and notifies them appropriately.

Event-Types are defined by creating an event calss which inherits from a `SCEvent`.

```
public class MyEvent extends SCEvent {

  public final static int EVENT_X = createNewEvent();
  public final static int EVENT_Y = createNewEvent();

  public final static int EVENT_X_FEATURE_A = 0;
  public final static int EVENT_X_FEATURE_B = 1;

  public MyEvent(Object source, int eventID, Object...objects ) {
    super(source, eventID, objects);
  }
}
```

In this example the event class *MyEvent* is defined. Several Event-Types are used in this event calss. Appropriate constansts are defined using `createNewEvent()`. In the above examples we have two types of events: X and Y

! The constants should not be created manually. `createNewEvent()` guarantees that each Event-Type obtains a unique ID.

The constructor of an event requires the source of the event to be passed to it as a parameter. Moreover the ID of the event type is required. In addition, arbitrarly many objects can passed to an event, they can contain information about the notification and such and additional user data.

Different user data which are passed to an event can only be seperated by their position in the sequence. Therefore a sequence should be handled carefully and the position of a particular piece of a user data should be defined as a constant. In our above example feature A of event X has the position 0 and feature B has the postition 1.

Creating Events An event is triggered by creating an Event instance and calling the operation `fireEvent()` of the `ServiceContext`.

```
this.serviceContext.fireEvent(new MyEvent(this, MyEvent.EVENT_X, object1, object2));
```

The first parameter of the Event-Constructor is the caller, therefore `this` must be specified here.

Registering a Listener A listener can be realized simply be implementing the interface `SCEventListener`.

```
public interface SCEventListener extends EventListener {
  public void eventDispatched(SCEvent event);
}
```

If an event occurs the opration `eventDispatched()` is called and the event is passed as parameter.

The listener is registered at the event context using the following operation:

```
public boolean addEventListener(Class<? extends SCEvent> eventtype,
                               SCEventListener listener){
  ...
}
```

One must specify which classes of events shall be listened. The listener is then notified for all types of events of this class.

Reacting to a Notification In order to react to a notification a listener must be registered at the `ServiceContext`. In the case of an event an `SCEvent`-Object is passed to the listener. The listener can query the type of the event and included user data from this object. A realization might look as follows:

```
public class MyListener implements SCEventListener {
  public void eventDispatched(SCEvent event) {
    if (event.getEventID==MyEvent.EVENT_X) {
      int featureA = event.getObject(MyEvent.EVENT_X_FEATURE_A, Integer.class);
      String featureB = event.getObject(MyEvent.EVENT_X_FEATURE_B, String.class);
      ...
    } else if ...
      ...
    }
  }
}
```

In this example we first check to see if an event of type X has occured, if so as a user data (in this example `int` and `String`) are queried in order to react on the event.