SiDiff 2.0 – Service-Howto

Sven Wenzel

26. Oktober 2009

Inhaltsverzeichnis

Dokumentation

Dieses Dokument wird fortlaufend gepflegt. Die nachfolgende Tabelle gibt eine Übersicht über die Änderungen in einzelnen Versionen.

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 Einleitung

1.1 Warum Services?

SiDiff ist nicht bloß ein Differenzalgorithmus, sondern ein kompletter Werkzeugkasten zum Bau verschiedener Differenzwerkzeuge. Zudem ist es kein Produkt, sondern Forschungsgegenstand. Um eine hohe Flexibilität zu erreichen, also einzelne Komponenten austauschen zu können, wurde SiDiff auf Basis von Services realisiert.

1.2 Grundidee

Die Grundidee ist, dass jede Funktionsgruppe, die SiDiff anbietet und die ggf. mal gegen eine andere Implementierung ausgetauscht werden könnte, als eigener Service realisiert wird. So gibt es z.B. jeweils eigene Services für:

- Einlesen von Dokumenten
- Annotieren mit künstlichen Attributen, wie z.B. Pfaden oder Metriken
- Ähnlichkeitsberechnung
- Matching
- ... (uvm.)

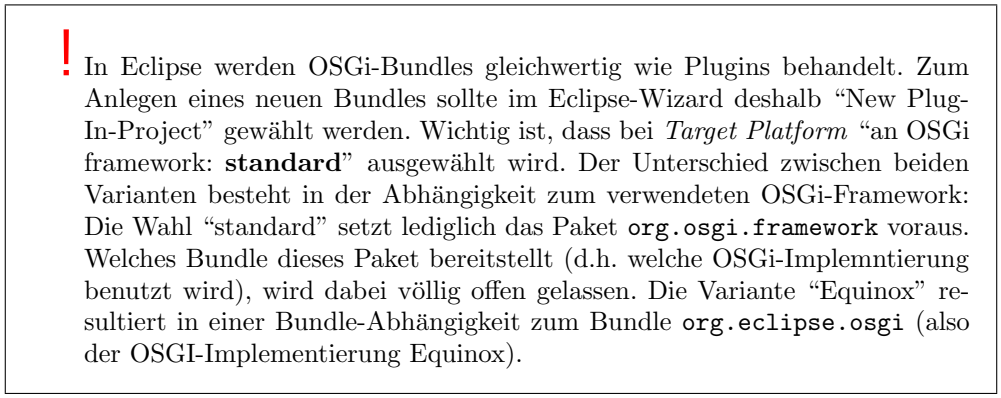
Zudem können sich Services auf mehrere Module verteilen. Das Annotieren z.B. unterteilt sich weiter in

- Erweiterung des Datenmodells, um Annotationen an Modellelemente anzuhängen,
- der Annotationsalgorithmus, der alle Modellelemente nacheinander abläuft
- und einzelne Annotationsfunktionen, die die künstlichen Attributwerte berechnen.

1.3 Technische Realisierung

Technisch werden die Services und Module durch OSGi-Bundles realisiert. Ferner benutzen wir Eclipse Equinox als OSGi-Plattform.

Ein *Bundle* ist i.W. eine Ansammlung von Java-Klassen und weiteren Ressourcen.



In Eclipse werden OSGi-Bundles gleichwertig wie Plugins behandelt. Zum Anlegen eines neuen Bundles sollte im Eclipse-Wizard deshalb "New Plug-In-Project" gewählt werden. Wichtig ist, dass bei *Target Platform* "an OSGi framework: **standard**" ausgewählt wird. Der Unterschied zwischen beiden Varianten besteht in der Abhängigkeit zum verwendeten OSGi-Framework: Die Wahl "standard" setzt lediglich das Paket `org.osgi.framework` voraus. Welches Bundle dieses Paket bereitstellt (d.h. welche OSGi-Implemntierung benutzt wird), wird dabei völlig offen gelassen. Die Variante "Equinox" resultiert in einer Bundle-Abhängigkeit zum Bundle `org.eclipse.osgi` (also der OSGI-Implementierung Equinox).

Ein *Service* ist eine Funktion, die bereitgestellt wird. Hierzu unterscheiden wir zwischen Servicedefinition (Schnittstelle) und Realisierung. Die Schnittstelle ist i.W. wirklich nur ein Java-Interface, während die Realisierung dieses Interface implementiert.

Für einen Service gibt es immer mindestens zwei Bundles. Ein Bundle stellt die Servicedefinition und bietet ggf. Klassen und Ressourcen an, die von allen Realisierungen genutzt werden können. Mindestens ein weiteres Bundle stellt eine Implementierung des Services bereit. Mehrere verschiedene Implementierungen sind selbstverständlich möglich.

1.4 Arten von Services

Wir unterscheiden folgende Arten von Services:

- einfache Services (siehe Kochrezept A)
- instanzierbare Services (siehe Kochrezept B)
- konfigurierbare Services (siehe Kochrezept C)
- konfigurierbare und instanzierbare Services (siehe Kochrezept D)

Als *einfachen Service* bezeichnen wir einen Service, der durch *ein Objekt* bereitgestellt wird, dass eine bestimmte Schnittstelle definiert.

Als *instanzierbaren Service* bezeichnen wir einen Service, der bei jeder Anforderung durch ein *neues Objekt* bereitgestellt wird. Das Konzept ähnelt dem OSGi-Konzept der `ServiceFactory`, ist jedoch grundsätzlich anders, da es bei *wirklich jeder* Anforderung ein neues Service-Objekt erzeugt – auch wenn der Service mehrfach aus dem gleichen Bundle angefordert wird. Um uns von der `ServiceFactory` abzugrenzen, wird hier mit einem `ServiceProvider` gearbeitet[1].

[1] Aus OSGi-Sicht ist der `ServiceProvider` ein Service, der nur ein Objekt erzeugt und zurückgibt. Wir bezeichnen aber dieses Objekt als Implementierung des Service, da der Mechanismus des `ServiceProviders` transparent erfolgt.

Konfigurierbare Services sind Services, die mittels geeigneter Daten (z.B. Konfigurationsdatei) auf spezielle Dokumenttypen angepasst werden können. Hier wird ein Konfigurator registriert, mit dessen Hilfe eine Service-Instanz erzeugt und konfiguriert werden kann. Im Anschluss ist diese Instanz dauerhaft konfiguriert und mehrfach verwendbar. Die Koexistenz verschieden konfigurierter Instanzen eines Service ist damit möglich.

Konfigurierbare und instanzierbare Services sind eine Mischform aus den beiden vorherigen Service-Typen. Die Services werden konfiguriert und bei jeder Anforderung wird ein neues Objekt bereitgestellt. Im Wesentlichen wird hier der `ServiceProvider` konfiguriert.

1.5 Hierarchien von Services

Services lassen sich hierarchisch klassifizieren. Erste Klassen von Services wurden bspw. bereits in Abschnitt 1.4 eingeführt. Das sich die Klassifizierung von Services hierarchisch strukturieren läßt ist bspw. daran erkennbar, dass jeder konfigurierbare Service natürlich auch ein einfacher Service mit zusätzlichen Eigenschaften ist. Ein weiteres Beispiel aus SiDiff ist der MatchingService, von dem es spezielle Varianten in Form des HashMatchingService und des IterativeMatchingService gibt.

Im Zuge des Service-Managements durch den ServiceHelper (s. Abschnitt 1.6) wird dieser Service-Hierarchie dadurch Rechnung getragen, dass jeder Service nicht nur unter seinem eigenen Service-Interface beim OSGI-Framework registriert wird, sondern auch unter der Schnittstelle seiner Superklassen. Der HashMatchingService wäre demnach also nicht nur als HashMatchingService, sondern auch als MatchingService registriert. Gleichzeit bekäme ein Client bei der Anforderung MatchingService tatsächlich einen Service zurück. In der Regel wird hier der am wenigsten spezielle Service vom ServiceHelper zurückgeliefert. Sind zwei oder mehr Service gleich speziell (d.h. sie besitzen die selbe Tiefe in der Sevice-Hierachie), so wird irgendeiner dieser Services zurückgegeben.

1.6 Der ServiceHelper

Der ServiceHelper kapselt die Zugriffe auf den Service-Layer des OSGi-Frameworks. Wir benutzen den ServiceHelper somit als allgemeine Schnittstelle für Aufgaben wie bspw. die Registrierung oder das Anfordern von Services. Die direkte Kommunikation mit dem OSGi-Framework über den OSGi-BundleContext sollte vermieden werden. Der ServiceHelper kapselt alle Zugriffe für eine bessere Wartbarkeit und die gleichartige Nutzung anderer Servicearten.

1.7 Literatur

Als Einführung in die OSGi-Thematik empfehlen wir:
Gerd Wütherich et.al.,
Die OSGi Service Platform,
dpunkt Verlag,

ISBN 978-3-89864-457-0,
2008

2 Konventionen

2.1 Trennung von Schnittstelle und Implementierung

Wie bereits oben erwähnt, gibt es für jeden Service mindestens zwei Bundles, damit Schnittstelle und Implementierung getrennt werden.

Es kann Ausnahmen geben, in denen Schnittstelle und Implementierung eines Service nicht getrennt werden. Dies ist insbesondere dann der Fall, wenn es nur eine Implementierung für den Service geben kann und die Möglichkeit zur Austauschbarkeit unsinnig ist. **Diese Ausnahmen sollten gemeinsam abgestimmt werden.**

2.2 Namensgebung

Jeder Service soll seinen eigenen Namensraum definieren, dazu wird das Bundle entsprechend benannt. Der Name beginnt immer mit `org.sidiff.`. Dann folgt ggf. ein `core.`, falls das Bundle dem SiDiff-Kern zuzuordnen ist. Im Anschluss folgt der eigentliche Bundlename.

Beispiele: Matching (gehört zum Kern) → `org.sidiff.core.matching`, Fingerprints (nicht Bestandteil des Kerns) → `org.sidiff.fingerprints`.

Hat ein Bundle mehrere Unterbundles oder gibt es für den Service verschiedene Realisierungen, so werden hierzu weitere Pakete angehängt.

Beispiel `org.sidiff.core.matching.idbased` und `org.sidiff.core.matching.iterative`

2.3 Sichtbarkeiten

OSGi bietet die Möglichkeit, in der Manifest-Datei eines Bundles festzulegen, welche Pakete nach aussen sichtbar sein sollen, und welche für den internen Gebrauch innerhalb des Bundles gedacht sind. Dieser Mechanismus ist bitte soweit möglich auszunutzen, sodass nur die Klassen nach aussen sichtbar sind, die dies wirklich erfordern. Um Klassen zu “verstecken” kann man ggf. ein Unterpaket `internal` anlegen, das man nach aussen nicht freigibt.

3 Kochrezept A: Anbieten und Nutzen eines einfachen Services

Zum Anbieten eines Service müssen zwei Bundles angelegt werden. Eines mit der Service-Schnittstelle und eines mit der Implementierung des Service.

3.1 Anlegen des Schnittstellen-Bundles

Das Schnittstellen-Bundle wird angelegt wie ein Bundle, das Klassen und Ressourcen anbietet (siehe WhitePaper "Introduction to the SiDiff 2.0 Development Environment").

Im Wesentlichen bietet dieses Bundle nämlich ein Java-Interface an. Optional können hier auch noch andere Klassen und Ressourcen angeboten werden. Zum Beispiel: Hilfsfunktionen, die von verschiedenen Implementierungen genutzt werden können.

Definition der Service-Schnittstelle Die Service-Schnittstelle ist ein reines Java-Interface, welches das Interface `org.sidiff.common.services.Service` erweitert. Hier sollten alle Methoden definiert werden, die der Service anbieten soll.

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

> ! Es sollen keine abstrakten Klassen als Service-Schnittstellen eingesetzt werden. Möchte man ein abstraktes Verhalten bereits anbieten, so kann zusätzlich zum Interface eine abstrakte Klassen bereitgestellt werden, von der konkrete Service-Implementierungen erben können.

3.2 Anlegen des Bundles mit der Implementierung

Auch das Bundle mit der Implementierung wird wie im WhitePaper "Introduction to the SiDiff 2.0 Development Environment"beschrieben erstellt, **jedoch mit einer wichtigen Änderung**:

In Schritt 3 wird ausgewählt, dass ein Activator generiert werden soll. Der vorgeschlagene Name sollte beibehalten werden.

Abhängigkeit zum Schnittstellen-Bundle Bevor der Service implementiert werden kann, ist im Manifest-Editor unter *Dependencies* das Schnittstellen-Bundle als *Required Plugin* hinzuzufügen. Im Anschluss können die Klassen und damit auch das Interface aus dem Schnittstellen-Bundle verwendet werden.

Implementieren des Services Um den Service zu implementieren, ist eine neue Klasse anzulegen, die das Interface aus dem Schnittstellen-Bundle implementiert. Hierzu können auch beliebig viele andere Klassen angelegt und verwendet werden. auch das Nutzen weiterer Services ist selbstverständlich möglich.

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

Bekanntmachen der Service-Implementierung mittels Activator Bevor ein konkreter Service genutzt werden kann, muss er registriert werden. Es muss also irgendwo gesagt werden "Hallo, ich bin eine Implementierung von Service X!". Dies erfolgt bei der zentralen OSGi-Verwaltung, die über den sogenannten BundleContext zugreifbar ist. Den BundleContext erhält man als Parameter in der Start-Methode des Activators. Hier wird auch die Service-Implementierung registriert.

> **!** Es werden nur Implementierungen von Services registriert. Die Service-Schnittstellen werden allen betroffenen als bekannt vorausgesetzt und nicht explizit registriert.

Hier ein Beispiel-Activator. Hinter `MyService.class` verbirgt sich ein Java-Interface, dass im Schnittstellen-Bundle definiert ist.

```
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 {
  }

}
```

Die `start()`-Methode wird vom OSGi-Framework automatisch aufgerufen, wenn das Bundle gestartet wird.

Mit `registerService()` wird der Service registriert. Hierzu werden folgende Parameter übergeben:

1. Der OSGI-BundleContext
2. Der Service, dessen Implementierung hier angeboten wird.
3. Eine **Instanz** der Service-Implementierung.
4. Der Dokumenttyp, den der Service unterstützt, oder `null`, falls der Service Dokumenttyp-unabhängig ist.
5. Eine optionale Variantenbezeichnung oder `null`.
6. Als fünfter, optionaler(!) Parameter kann noch, wie bei allen OSGI-Services, ein Dictionary mit Schlüssel-Wert-Paaren angegeben werden, das in der OSGi-ServiceRegistry gespeichert wird.

Falls notwendig, können in der `stop()`-Methode Anweisungen angegeben werden, die auszuführen sind, falls das Bundle aus dem System herausgenommen wird.

> ! Per Konvention sollten alle Bundles ihren Namen, der gleichzeitig Projektname ist, auch als String-Konstante speichern, um darauf referenzieren zu können. Die Konstante sollte nach Möglichkeit im Activator definiert werden, um sie innerhalb eines Projektes schnell zu finden.

3.3 Verwenden des Services

Der oben implementierte Service kann nun in anderen Bundles verwendet werden. Hierzu muss im Manifest-Editor des benutzenden Bundles das Bundle mit der Service-Schnittstelle als *Required Plugin* in den *Dependencies* eingetragen werden. Die Registrierung des implementierenden Bundles ist nicht notwendig und sollte zwecks Aufrechterhaltung der Austauschbarkeit von Implementierungen auf keinen Fall eingetragen werden.[2]

Zudem muss das Bundle `org.sidiff.common.services` als *Required Plugin* eingetragen werden. Es bietet Hilfsfunktionen wie den `ServiceHelper` an.

Im benutzenden Bundle kann die Implementierung des Service wie folgt angefordert und benutzt werden:

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

[2] Welche Implementierung gerade zur Verfügung steht ergibt sich aus der aktuellen Laufzeitumgebung. In Eclipse-Runtime-Umgebungen z.B. zum Debuggen, wird dies durch Auswahl aller zu ladenden Plugins/Bundles festgelegt.

`MyService` stellt dabei das Java-Interface dar, das die Service-Schnittstelle definiert. Über die konkrete Implementierung wird hier nichts bekannt. `context` ist der BundleContext von OSGi. Der zweite Parameter ist die Schnittstelle des angeforderten Service. Der dritte Parameter kann einen Dokumenttyp vorgeben für den der Service angefordert wird. Der vierte Parameter kann eine Variante vorschreiben. Im einfachen Fall können der dritte und vierte Paramter `null` sein.

Es ist zu beachten, dass bei dem hier vorgestellten Verfahren, *ein Objekt* als Serviceimplementierung registriert wurde. Alle Nutzer des Services bekommen also Zugriff auf dasselbe Objekt. Dies sollte bei der Definition der Schnittstelle und bei der Implementierung (Stichwort temporäre Daten) berücksichtigt werden.

4 Kochrezept B: Definition und Verwendung eines ProvideableService

Es kann oft vorkommen, dass ein Service in verschiedenen Kontexten genutzt werden soll; und das ggf. gleichzeitig. Deshalb ist es nicht sinnvoll, wenn genau eine Service-Instanz (also ein Objekt) beim OSGi-Framework registriert wird. Mit `ProvideableService` und `ServiceProvider` bieten wir ein Konzept an, um Service-Instanzen erst dann anzulegen wenn der Service angefordert wird, und außerdem um jedem Nutzer des Services eine neue Instanz anzubieten. Die Klassen werden beide im Bundle `org.sidiff.common.services` bereitgestellt.

> ! OSGi bietet eine sogenannte `ServiceFactory` an. Es handelt sich dabei um ein Interface, welches ein Service implementieren kann. Ist dies der Fall wird, beim Anfordern des Services aus verschiedenen Bundles die Factory verwendet und jedem Bundle ein neues Objekt zurückgegeben. Da die Service-Instanzen, die mittels `ServiceFactory` erzeugt werden, jedoch transparent vom OSGi-Framework pro Bundle zwischengespeichert werden, wird zwei unterschiedlichen Instanzen eines Services der gleiche Service angeboten. Nebenläufigkeit und Trennung von Informationen sind somit nicht mehr möglich. Das Konzept der `ServiceFactory` wird deshalb bei SiDiff nicht verwendet.

4.1 Definition der Service-Schnittstelle

Die Definition der Service-Schnittstelle erfolgt wie bei einfachen Services, jedoch muss das Interface zusätzlich vom Interface `ProvideableService` erben. Dieses Interface ist leer und dient als “Marker” im automatischen Instanzierungsprozess.

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

Außerdem wird die Definition eines `ServiceProviders` benötigt, welcher die Methode `createInstance()` definiert.

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

Wichtig ist, dass das Interface folgender **Konvention** folgt:

1. Es heißt genauso wie die Service-Schnittstelle trägt das Suffix `Provider`.
2. Es liegt im selben Paket wie die Service-Schnittstelle.
3. Es erbt vom Interface `ServiceProvider`, getypt auf die Service-Schnittstelle.

4.2 Implementierung des Services

Der Service selbst wird wie gewohnt realisiert, indem er die Schnittstelle implementiert.

Zusätzlich ist nun die Implementierung des ServiceProviders notwendig. Dieser verfügt über eine `createInstance`-Methode, die ausgerufen wird, um eine Instanz des Service zu erzeugen.

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

Die Methode muss ein Objekt vom Typ des entsprechenden Service zurückgeben.

4.3 Bekanntmachen der Service-Implementierung

Die Bekanntmachung der Service-Implementierung erfolgt wieder über den Activator. Jedoch wird diesmal der `ServiceProvider` registriert. Die weiteren Parameter sind analog zur Registrierung einfacher Services.

```
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 {
  }

}
```

Die Service-Implementierung selbst muss nicht registriert werden. Sie wird über den `ServiceProvider` bereitgestellt.

4.4 Verwenden des Services

Der oben implementierte Service kann nun genauso in anderen Bundles verwendet werden wie einfache Services. Hierzu muss wieder das Schnittstellen-Bundle als Abhängigkeit im Manifest-Editor eingetragen werden. Zudem besteht wieder die Abhängigkeit zum Bundle `org.sidiff.common.services`.

Der Zugriff auf den `ServiceProvider` erfolgt transparent über den `ServiceHelper`. Im benutzenden Bundle kann die Implementierung des Service also wie bei einfachen Services angefordert und benutzt werden:

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

`MyService` stellt dabei wieder das Java-Interface dar, das die Service-Schnittstelle definiert. Über den eingesetzten `ServiceProvider` ist hier also nichts bekannt. Der Zugriff auf den Provider und die Instanzierung der eigentlichen Service-Implementierung erfolgt transparent.

Der `ServiceHelper` prüft, ob der angeforderte Service ein `ProvideableService` ist. Wenn ja, wird beim OSGi-Framework ein entsprechend benannter `ServiceProvider` (hier `MyServiceProvider`) gesucht und dessen `createInstance()`-Methode aufgerufen. Der Nutzer des Services bekommt durch die Kapselung im `ServiceHelper` jedoch nichts davon mit.

> **!** Es ist zu beachten, dass bei dem hier vorgestellten Verfahren, *immer* ein *neues* Objekt als Service-Implementierung zurückgeliefert wird. Wird daselbe Objekt mehrfach benötigt, sollte also entsprechend nach dem ersten Anfordern eine Referenz darauf angelegt werden.

5 Kochrezept C: Konfigurierbare Services

In viel Fällen benötigen wir Services, die z.B. für einen bestimmten Dokumenttyp konfiguriert werden. Zudem möchten wir unterstützen, dass mehrere, verschieden konfigurierte Services eines Typs parallel existieren.

Wir ermöglichen die Koexistenz von mehreren Service-Instanzen, die verschieden konfiguriert sind. Die Instanzen unterschieden wir anhand des Dokumenttyps für den sie bestimmt sind. Da SiDiff darauf ausgelegt ist, Dokumente unterschiedlichen Typs zu verarbeiten und die Konfiguration sich insbesondere von Dokumenttyp zu Dokumenttyp unterscheidet, benutzen wir den Dokumenttyp als Schlüsselattribut.

Um zudem auch für einen Dokumenttyp noch mehrere Konfigurationen zuzulassen, kann zusätzlich zum Dokumenttyp auch noch eine Variante angegeben werden. Diese ist ein einfacher String, um verschiedene Konfigurationen zu unterscheiden.

Services müssen nur einmal konfiguriert werden und können danach beliebig oft verwendet werden. Dies ermöglicht eine effiziente Integration in Algorithmen, da u.U. aufwendige Initialisierungen nur einmal auszuführen sind.

5.1 Definition eines konfigurierbaren Services

Service-Schnittstelle Um einen konfigurierbaren Service zu definieren, muss die Service-Schnittstelle lediglich das Interface `ConfigurableService` erweitern. Es befindet sich, wie die anderen Service-relevanten Teile, im Bundle `org.sidiff.common.services`

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

Die `configure()`-Methode wird aufgerufen, um den Service mit beliebigen Daten (`Object...`) zu konfigurieren. Es kann sich hierbei um eine freidefinierbare Menge von Objekten, wie z.B. String oder Konfigurationsdateien handeln. Beim registrieren des Services müssen diese Daten entsprechend übergeben werden (s.u.). Der Aufruf der Methode `configure()` des konfigurierbaren Services geschieht hierbei für den Nutzer des Services völlig transparent.

Der Rückgabewert der `configure()`-Methode ist der Dokumenttyp für den die Konfiguration geeignet ist. Da sich der Typ i.d.R. aus den Konfigurationsdaten ableitet, kann er sinnvollerweise nur hier ermittelt werden. Mit dem hier zurückgegebenen Typ als Schlüssel wird der Service später im OSGi-Framework registriert.

Die `deconfigure()`-Methode kann optional dazu verwendet werden, Konfigrationen zu deinitialisieren. Sie wird aufgerufen, wenn der zuvor konfigurierte Service deregistriert wird.

Mit `getProperties()` kann ein String-zu-String-Dictionary mit weiteren Service-Eigenschaften als Schlüssel-Wert-Paare zurückgeliefert werden. Bei der Registrierung des Services wird das Dictionary wird (nach der Ausführung von `configure()`) abgefragt und an das OSGi-Framework weitergegeben. Im Normalfall werden keine weiteren Eigenschaften definiert und es kann `null` zurückgegeben werden.

Da der konfigurierte Service automatisiert instanziert wird, benötigt er einen parameterlosen Konstruktor.

Implementierung Die Implementierung des Service erfolgt wie bei einfachen Services. Sie unterscheidet sich nur durch die zusätzlichen Methoden des `ConfigurableService` womit der ServiceInstanz die Konfiguration übergeben wird.

Gehen wir davon aus, dass wir bereits eine Schnittstelle Namens `MyConfigurableService` erstellt haben. Diese solle wie oben beschrieben von `ConfigurableService` abgeleitet sein und keine zusätzlichen Methoden definieren. Eine Mögliche Implementierung könnte folgendermaßen aussehen:

```
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 Bekanntmachen des konfigurierbaren Services

Der konfigurierbare Service wird angeboten, indem die Schnittstelle und die konkrete Implementierung des konfigurierbaren Services beim ServiceHelper registriert werden. Dies erfolgt i.d.R. im 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 Verwenden eines konfigurierbaren Services

Die Verwendung eines konfigurierbaren Services erfolgt in zwei Schritten. Im ersten Schritt wird der Service konfiguriert. Dies braucht nur einmal gemacht werden. Im zweiten Schritt wird die konfigurierte Instanz des Service benutzt. Dies kann beliebig oft erfolgen.

Service konfigurieren Die Konfiguration des Services erfolgt auch mit dem `ServiceHelper`, über folgende Methode:

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

Diese wird bspw. folgendermaßen aufgerufen:

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

Als Parameter werden übergeben: der BundleContext, die Service-Schnittstelle, der Dokumenttyp, ggf. die Variantenbezeichnung, und eine beliebige Anzahl weiterer Objekte als Konfigurationsdaten. Es empfiehlt sich, eine Konfigurationsdatei zu übergeben.

Nach diesem Aufruf steht der Service in konfigurierter Form zur Verfügung. Die Eigentliche Konfiguration erfolgt transparent durch den `ServiceHelper`.

Service anfordern Um einen fertig konfigurierten Service zu verwenden, wird dieser wie alle anderen Services über den `ServiceHelper` angefordert. Hier müssen jedoch der Dokumenttyp oder der Dokumenttyp und die Variante als Parameter mitgegeben werden.

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

oder

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

! Innerhalb des SiDiff-Projektes verwenden wir EMF-Modelle, daher bietet sich grundsätzlich an, den Namespace-URI des jeweiligen Metamodells als Dokumenttyp zu verwenden. (siehe Beispiel)

6 Kochrezept D: Konfigurierbare und instanzierbare Services

Instanzierbare Services können auch als konfigurierbare Services mit zusätzlichen Konfigurationsdaten eingestellt werden. Hierzu wird wieder ein `ServiceProvider` angelegt, der jedoch auch ein `ConfigurableService` ist. Um das Konzept einfach umzusetzen, werden spezielle Java-Interfaces angeboten, die beide Mechanismen miteinander vereinen.

Der `ServiceProvider` kann dann mit unterschiedlichen Daten konfiguriert werden. Die Koexistenz verschieden konfigurierter Provider ist ebenfalls wieder möglich. Wenn der konkrete Service angefordert wird, wird jedesmal eine neue Instanz zurückgegeben. Ein wiederholtes Einlesen einer Konfiguration ist dabei nicht mehr notwendig.

6.1 Definition der konfigurierbaren und instanzierbaren Services

Service-Schnittstelle Der neue Service muss das Interface `ConfigurableProvideableService`, welches lediglich wieder ein Marker-Interface darstellt, implementieren. Dieses dient zur Markierung, damit die Anforderung des Service transparent zum konfigurierten `ServiceProvider` delegiert werden kann.

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

Zudem benötigt man einen `ServiceProvider`. Hier wird diesmal jedoch ein konfigurierbarer Provider benötigt, welcher mit dem Interface `ConfigurableServiceProvider` erstellt werden kann.

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

Auch hier gilt die Namenskonvention, dass der Provider genauso benannt ist wie der Service und zusätzlich das Suffix `Provider` trägt.

Implementierung des Service Die Implementierung des Service erfolgt hier wie bei einfachen Services. Ein besonderes Verhalten muss nicht berücksichtigt werden. Im Gegenteil: anders als beim konfigurierbaren Service sind hier keine zusätzlichen Methoden zu Konfiguration zu implementieren. Die Konfiguration betrifft diesmal den `ServiceProvider`.

Implementierung des Providers Der `ServiceProvider` ist dafür zuständig mit jeder Anforderung des Services eine neue Instanz anzulegen und zurückzugeben. Zudem soll er mit Hilfe von Konfigurationsdaten konfiguriert werden können. Durch die Implementierung des zuvor definierten Provider-Interfaces muss der Provider auch die Schnitt-

stelle des `ConfigurableServiceProvider`s bedienen, welche wiederum `ServiceProvider` und `ConfigurableService` vereint.

```
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;
  }

}
```

Die Methoden `configure()`, `deconfigure()` und `getProperties()` sind zur Realisierung der Konfigurierbarkeit gedacht. Die benötigten Informationen für eine Konfiguration sollten beim Aufruf der `configure()`-Methode gespeichert werden (hier angedeutet durch `MyConfiguration`). Wie sich die Konfiguration bis zur Instanzierung des Service gemerkt wird ist egal. Es wäre auch möglich, alle übergebenen Objekte (`Object... configData`) zwischenzuspeichern und erst bei der Instanzierung komplett auszuwerten. **Es ist jedoch darauf zu achten, dass `configure()` den gültigen Dokumenttyp, für den die Konfiguration gedacht ist, zurückgibt.**

Die `createInstance()`-Methode wird aufgerufen, wenn der Service angefordert wird. Hier ist also die konkrete Instanz des Service zu erzeugen, ggf. zu konfigurieren (hier angedeutet durch `ms.setConfiguration(config)`, alternativ könnte je nach Konfiguration eine andere Instanz angelegt werden), und zurückzugeben.

Da der konfigurierte `ServiceProvider` automatisiert instanziert wird, benötigt er einen parameterlosen Konstruktor.

6.2 Bekanntmachen des Services

Der konfigurierbare und instanzierbare Service wird angeboten, indem die Schnittstelle und die konkrete Implementierung des Services beim ServiceHelper registriert werden. Dies erfolgt i.d.R. im 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 Verwenden des Services

Auch die Verwendung des konfigurierbaren und instanzierbaren Service funktioniert wie bei einem normalen konfigurierbaren Service.

Service konfigurieren Die Konfiguration des Services erfolgt wieder mit dem `ServiceHelper`, über folgende Methode:

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

Diese wird bspw. folgendermaßen aufgerufen:

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

Service verwenden Um einen fertig konfigurierten, instanzierbaren Service zu verwenden, wird dieser wie alle anderen Services über den `ServiceHelper` angefordert.

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

oder

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

Mit dem Aufruf von `getService()` wird der `ConfigurableServiceProvider` mittels `createInstance()`-Methode aufgefordert, eine konkrete Instanz des Service zu erzeugen und zurückzugeben. Mit jedem Aufruf von `getService()` wird also eine neue Instanz des konfigurierten Service zurückgegeben. Sowohl die eigentliche Konfiguration des Services als auch dessen Instanzieerung erfolgen für den Nutzer wieder völlig transparent.

7 Zusammenspiel von Services: ServiceContext und Kommunikation

Es gibt einige Services in SiDiff, die eng miteinander verwoben sind. Das OSGi-Framework erlaubt uns, Services nach Belieben bereitzustellen, Implementierungen auszutauschen und Dinge zu kapseln. Doch manchmal haben wir genau den Fall, dass mehrere Service-Instanzen gemeinsam einen Kontext bilden.

Hierzu haben wir den `ServiceContext` entwickelt. Er ermöglicht, mehrere Services zusammenzustecken, die gegenseitig ihren Kontext bilden, und bietet den Services an, auf die anderen Services eben dieses Kontextes zuzugreifen. Zudem bietet er Mechanismen zur Kommunikation zwischen den Services. Er stellt einen Eventbus bereit, über den die Services Nachrichten verschicken können bzw. auf dem sie lauschen können.

Ein mögliches Beispiel für sogenannte Kontext-sensitive Services in SiDiff sind die Korrespondenz- und die Kandidatenverwaltung. Wann immer eine neue Korrespondenz angelegt wird, stehen die beteiligten Elemente nicht mehr als Kandidaten zur Verfügung.

7.1 Kontext und Kontext-sensitive Services

Der Kontext wird durch die Klasse `ServiceContext` realisiert. Hier können verschiedene Services hinzugefügt werden.

```
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) {
  ...
}
```

Mit `putService()` kann eine Service-Instanz `service` in den Kontext eingefügt werden. Die Service-Schnittstelle `serviceID` dient dabei zur Identifikation einer Service-Instanz. Pro Schnittstelle kann im Kontext immer nur eine Instanz existieren.

`getService()` ermöglicht, bestimmte Services abzufragen. Vorher sollte aber mit `containsService()` geprüft werden, ob der entsprechende Service auch im Kontext vorhanden ist, da es sonst zu Exceptions kommen kann.

Nachdem alle Services in einen Kontext eingefügt wurden, ist dieser mit `initialize()` zu initialisieren. Hierbei können Parameter-Objekte mitgegeben werden, die an alle

Services im Kontext weitergereicht werden. Ein Beispiel bilden die beiden Modellressourcen eines Vergleichs. Sollen einige der Services nicht alle Parameter-Objekte übergeben bekommen, so kann hierzu beim Einfügen des Services (mit `putService()`) eine optionale Liste von Indizes angegeben werden. Die Indizes beschreiben, welche der Parameter-Objekte von `initialize()` an den jeweiligen Service weitergereicht werden. Mit `setDefaultParams()` kann eine Liste von Indizes vorgegeben werden, die genutzt werden, falls `putService()` ohne Indizes aufgerufen wird.

> **!** Ein einmal initialisierter Kontext darf nicht mehr verändert werden. Das Einfügen oder Löschen weiterer Services führt zu Exceptions!

Kontext-sensitive Services Manche Services benötigen zu Beginn eine Initialisierung mit bestimmten Daten, die letztlich auch den Kontext bestimmen. Dies kann z.B. eine Menge von Dokumenten sein, die verglichen werden. Um sicherzustellen, dass alle beteiligten Services mit den gleichen Daten initialisiert werden, wird die Initialisierung vom `ServiceContext` mittels `initialize()` organisiert (siehe oben).

Services, die mit dem Kontext initialisiert werden sollen, müssen hierzu das Interface `ContextSensitiveService` implementieren.

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

Mithilfe von `initialize()` werden dem betreffenden Service der Kontext und ggf. weitere Objekte zur Initialisierung übergeben.

> **!** Es ist nicht notwendig, dass alle Services, die in einen `ServiceContext` eingefügt werden `ContextSensitiveService` implementieren. Dies ist nur notwendig, wenn diese Services selbst auf den Kontext zugreifen wollen.

Beispiele Im nachfolgenden Beispiel werden die Services A und B in einen Kontext eingefügt. Anschließend wird der Kontext mit zwei Modellen initialisiert. Da A und B jeweils die gleichen Initialisierungs-Parameter erwarten können Indizes weggelassen werden.

```
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);
```

Das Beispiel könnte durch das Setzen von Standard-Indizes (vor `initialize()`) ergänzt werden. Standardmäßig werden immer alle Parameter-Objekte von `initialize()` an die Services weitergegeben.

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

Hätten wir nun einen weiteren Service C, der nur das erste Modell und einen String als Parameter erwartet, sähe das Beispiel so aus:

```
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");
```

Eine Beispielrealisierung für Service A, die den Kontext benutzt um auf den anderen Service zuzugreifen, kann folgendermaßen aussehen.

```
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 Kommunikation zwischen Services

Services, die gemeinsam im Kontext stehen, bedürfen oft auch einen Informationsaustausch. Hierzu verwaltet der `ServiceContext` verschiedene `EventDispatcher`. Die jeweils das Observer-Muster realisieren. Jeder Dispatcher kümmert sich um die Verwaltung und Benachrichtung der Listener (Observer) eines bestimmten Event-Typs.

Event-Typen werden definiert indem eine Eventklasse angelegt wird, die von `SCEvent` erbt.

```
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 diesem Beispiel wird die Eventklasse *MyEvent* definiert. Um mehrere Event-Typen innerhalb dieser Eventklasse zu definieren, werden mit `createNewEvent()` Konstanten definiert. Im obigen Beispiel gibt es zwei Arten von `MyEvents`: X und Y.

> **!** Die Konstanten sollten nicht von Hand vergeben werden. `createNewEvent()` stellt sicher, dass jeder Eventtyp eine eigene ID erhält.

Der Konstruktor eines Events sieht vor, dass immer die Quelle eines Events übergeben wird. Zudem wird die ID des Eventtyps mitgeteilt. Zudem können einem Event immer beliebig viele Objekte übergeben werden, die neben der reinen Notifikation über aufgetretene Ereignisse auch Nutzdaten übertragen können.

Um die Nutzdaten, die der Notifikation über ein Ereignis beigefügt werden, besser zu unterscheiden, sollte die Reihenfolge, in der die Nutzdaten übergeben werden, klar definiert sein. Die Position einer bestimmten Nutzdateninformation sollte daher als Konstante definiert werden. Im Beispiel sind dies im Fall von Event X das Feature A an erster Position (=0) und Feature B an zweiter Position (=1).

Erzeugen von Events Ausgelöst wird ein Event durch erzeugen einer Event-Instanz und Aufruf der Methode `fireEvent()` des `ServiceContext`.

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

Der erste Parameter des Event-Konstrukturs ist der Aufrufer, hier muss also `this` übergeben werden.

Registrieren eines Listeners Ein Listener kann einfach realisiert werden, indem das Interface `SCEventListener` implementiert wird.

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

Im Fall eines Events wird die `eventDispatched()`-Methode aufgerufen und das Event als Parameter übergeben.

Der Listener wird beim `ServiceContext` mittels.

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

registriert. Hierzu muss angegeben werden auf welche Klasse von Events gehorcht werden soll. Der Listener wird dann für alle Typen von Events dieser Klasse benachrichtigt.

Reagieren auf Nachrichten Um auf eine bestimmte Nachricht zu reagieren, muss der Listener selbstverständlich beim `ServiceContext` registriert sein. Da dem Listener im Fall eines Events ein `SCEvent`-Objekt übergeben wird, können der Typ eines Ereignisses sowie die mitgeschickten Nutzdaten abgefragt werden. Eine Realisierung könnte wie folgt aussehen:

```
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 ...
      ...
    }
  }
}
```

Im Beispiel wird zunächst geprüft, ob Ereignis vom Typ X aufgetreten ist. Wenn ja, werden die Nutzdaten (hier ein `int` und ein `String`) abgefragt, um auf das Event zu reagieren.