

FirstSpirit™

Your Content Integration Platform

Entwicklerhandbuch für Komponenten FirstSpirit Version 4.2R4

Version	1.54
Status	in process
Datum	2011-08-29
Abteilung	FS-Core
Autor/ Autoren	A. Pfeiler
Copyright	2011 e-Spirit AG

MODDEV42R4DE_FirstSpirit_ModulDeveloperDoc

e-Spirit AG
Barcelonaweg 14
44269 Dortmund | Germany

T +49 231 . 286 61-30
F +49 231 . 286 61-59

info@e-spirit.de
www.e-spirit.de

e-Spirit^{AG}

Inhaltsverzeichnis

1	Thema dieser Dokumentation.....	7
2	FirstSpirit Modul-Grundkonzeption	8
2.1	Installation von Modulen	8
2.1.1	Validierung von Modulnamen bei der Installation bzw. Aktualisierung	8
2.1.2	Konfiguration von Modul-Komponenten	9
2.1.3	Aktivierung von Modul-Komponenten	9
2.2	Aktualisierung eines Moduls	9
2.3	Deinstallation eines Moduls	10
2.4	Import und Export von Modulen.....	10
2.5	Modul-Bestandteile	11
2.5.1	Ressourcen	11
2.6	Modul-Ereignisbehandlung.....	14
2.7	Modul-, Komponenten-Verzeichnisstruktur	14
2.8	FSM-Archiv-Struktur	17
2.9	Komponenten	18
2.9.1	Komponenten-Typen.....	18
2.9.2	Sichtbarkeit von Komponenten	25
2.9.3	Konfiguration von Komponenten	26
2.10	Classloading	28
2.10.1	Hierarchie.....	28
2.10.2	Classloading in Webanwendungen am Beispiel FirstSpirit u. WebSphere.....	30
2.11	Komponenten-Konfigurationsdateien.....	31



2.12 Logging in FirstSpirit.....	32
2.12.1 Globales Logging – fs-server.log / fs-client.log	32
2.12.2 Lokales Logging auf Modulebene	32
3 Das FirstSpirit 5 Modul- / Komponenten-Modell	33
3.1 Übersicht.....	33
3.2 Restrukturierung mit FirstSpirit 5.....	34
3.2.1 Änderungen zur FirstSpirit Version 4	34
3.2.2 Zielsetzung: Einfach, effizient und erweiterbar.....	35
3.3 Komponenten-Container (Typen).....	35
3.4 Initialisierung von Komponenten	36
3.5 Entladen von Komponenten.....	37
3.6 Event-Methoden von Komponenten.....	37
3.7 Modul Datei-, Archiv-Format (.fsm).....	39
3.8 Der Modul-Deskriptor	39
3.8.1 Beispiel Modul-Deskriptor	39
3.8.2 Modul-Deskriptor Tags und Attribute.....	40
3.9 Der Komponenten- <i><components></i> -Deskriptor-Teil	41
3.9.1 Komponenten-Deskriptoren und spezielle Eigenschaften.....	41
3.10 Getting started – Erste Schritte zur Modulentwicklung.....	53
3.10.1 FirstSpirit Version 4.2R4	53
3.10.2 Einrichten der IDE	54
3.11 GOM – FirstSpirit GUI Object Model	56
3.11.1 Typisierung und Mapping.....	57
3.11.2 Annotationen.....	62
3.11.3 Abstrakte GOM-Klassen.....	63
3.12 Von Gadgets, Aspects, Brokern und Agents	69
3.12.1 Gadgets	69



3.12.2 Aspekte (SwingGadget).....	69
3.12.3 Agents.....	95
3.12.4 Broker.....	95
3.12.5 SwingGadgetContext	96
3.12.6 Wertespeicherung (EditorValue, ValueEngineer).....	97
3.12.7 Aspekte (ValueEngineer)	102
3.12.8 Arbeiten mit Referenzen.....	109
3.13 FirstSpirit Security Architektur – Java Web Start (javaws).....	121
3.13.1 Verwendung des FirstSpirit Security-Managers/Classloaders	124
3.13.2 Zertifikat für Testzwecke erzeugen	125
3.14 Beispiel: Implementierung einer Eingabekomponente	126
3.14.1 Übersicht	126
3.14.2 GomForm - XML-Repräsentation im JavaClient.....	127
3.14.3 SwingGadgetFactory - Erzeugen eines typisierten Gadgets	129
3.14.4 SwingGadgets und die Verwendung von Standard-Aspekten.....	130
3.14.5 NotifyValueChange - Änderungen propagieren	133
3.14.6 ValueEngineerFactory.....	136
3.14.7 ValueEngineer - Werte eines SwingGadgets behandeln.....	137
3.14.8 Content-Highlighting für eine Komponente aktivieren.....	142
3.14.9 Aspekt Highlightable - Treffermarkierung für die Suche.....	142
3.14.10 Modul- und Komponenten-Deskriptor	145
3.14.11 Ant build.xml – Komplettes Beispiel	147
3.15 Komponentenlose Modul-Implementierung (JDBC-Connector-Modul)	149
3.16 Implementierung einer Bibliothek- (Library) Komponente (JDBC- Treiber-Modul).....	153
3.17 Modul-Implementierung mit den Komponenten-Typen – PUBLIC, SERVICE, LIBRARY	155
3.17.1 Modul-Komponenten und -Konfiguration	155
3.17.2 Schematische Darstellung des Moduls innerhalb von FirstSpirit	157



3.17.3 Der VScan-Module-Deskriptor.....	157
3.17.4 Vollständiger Modul-Deskriptor mit Drei Komponenten-Typen ...	158
3.17.5 Implementierung der SERVICE Basis-Komponente	160
3.17.6 Die PUBLIC-Komponente des Moduls.....	180
3.17.7 Die LIBRARY-Komponente(n) des Moduls	182
3.17.8 Konfiguration und Persistenz – fs-vsca.conf.....	185
3.18 Beispiel: Modul-Implementierung einer Komponente vom Typ WebApp.....	186
3.18.1 Entwicklung der Webapplikation	186
3.18.2 FirstSpirit spezifische Klassen.....	191
3.18.3 Anmerkungen zum Ant-Build-File.....	200
3.18.4 Installation des Modules.....	200
3.19 Namensgleichheit bei Modul-Ressourcen (z.B. Zip-Exportdateien)....	201
3.20 Erzeugung von FSM-Archiven (.fsm).....	203
3.21 Signieren von Modulen – Jar-Archiv-Klassen.....	203
3.22 Internationalisierung von Modulen – i18n	204
3.23 Icon Ressourcen.....	209
3.24 Text Ressourcen.....	209
3.25 Integration von Eingabekomponenten (Editoren) in den JavaClient ...	209
3.25.1 Beispiel Formular-Element (GOM-Form)	209
3.26 Modulansicht in der Server -und Projektkonfiguration	210
3.27 Ansicht Komponente Formular und Ausgabe-Kanäle (bspw. HTML, PDF).....	212
4 Listings	213
5 Klassen-, Interface-Beschreibungen	214
6 Abbildungsverzeichnis	215



7	Referenzen.....	217
8	Index.....	218



1 Thema dieser Dokumentation

Diese Dokumentation soll die Modul-/Komponenten-Entwicklung für FirstSpirit unterstützen und einen Einblick in die Konzeption geben. Es beschreibt die grundlegenden Designprinzipien, die Modul-Descriptor-Syntax und die Integration in FirstSpirit. Es enthält Beispiel-Implementierungen mit den wichtigsten Codeblöcken der in FirstSpirit möglichen Komponenten-Typen (siehe Kapitel 2.9.1 Seite 18).

Kapitel 2 erläutert die Verwendung von Modulen und Komponenten in FirstSpirit inklusive Installation und Deinstallation, Konfiguration, Aktivierung und Aktualisierung, Import und Export von Modulen. Darüber hinaus werden die Bestandteile von Modulen und verschiedene Komponenten-Typen vorgestellt (siehe Kapitel 2 ab Seite 8).

In **Kapitel 3** werden Komponenten-Typen näher beschrieben (siehe Kapitel 3 ab Seite 33). Die **Kapitel 3.7, 3.8, 3.9** und **3.20** beschäftigen sich mit FSM-Archiven, dem Modul- und dem Komponenten-Descriptor. **Kapitel 3.11** führt in das FirstSpirit GUI Object Model (GOM) ein, **Kapitel 3.13** geht auf Sicherheitsaspekte ein, **Kapitel 3.14** gibt ein Beispiel für eine einfache Editor-Komponente. Die **Kapitel 3.21** und **3.22** beschäftigen sich mit dem Signieren und Lokalisieren von Modulen.

Kapitel 4: Verzeichnis der in dieser Dokumentation verwendeten Listings (Seite 213)

Kapitel 5: Verzeichnis der Klassen- und Interface-Beschreibungen (Seite 214)

Kapitel 6: Abbildungsverzeichnis (Seite 215)

Kapitel 7: Referenzverzeichnis (Seite 217)

Kapitel 8: Index (Seite 218)



2 FirstSpirit Modul-Grundkonzeption

In FirstSpirit wird das Handling von Modulen/Komponenten neu organisiert. Zentrale Einheit ist das Modul, welches wiederum aus einer oder mehreren Komponenten besteht. Jede Komponente nutzt Ressourcen (shared, web-local – z.B. Klassen, Jar-Archiv, Properties-Dateien), hat eine bestimmte Sichtbarkeit (siehe Kapitel 2.9.2 Seite 25), besitzt einen Typ (siehe Kapitel 2.9.1 Seite 18) und Eigenschaften (siehe Kapitel 3.9.1 Seite 41). Es gibt drei Sichtbarkeitsstufen und sieben verschiedene Komponenten-Typen, die wiederum miteinander in einem Modul kombiniert werden können.

Jedes Modul enthält eine Descriptor-Datei (module.xml, siehe Kapitel 3.8 Seite 39), die alle Komponenten-Ressourcen, Sichtbarkeits-Stufen, Versionen einzelner Komponenten und Ressourcen sowie deren Abhängigkeiten, Abhängigkeiten zu anderen Modulen oder Modul-Herstellern, Modul- sowie Komponenten-Namen beschreibt. Jedes FirstSpirit-Modul-Archiv hat die Dateinamen-Erweiterung `*.fsm` (siehe Kapitel 3.7 Seite 39).

2.1 Installation von Modulen

Ausgangspunkt für die FirstSpirit Modul-Installation ist die Server- und Projektkonfiguration. Über diese wird das Modul (als FSM-Archiv) auf dem FirstSpirit-Server installiert. Die Ressourcen der system-weiten Komponenten stehen über den Server-Classloader (siehe Kapitel 2.10 Seite 28) ab diesem Zeitpunkt zur Verfügung. Es gibt *system-weite*, *web-lokale* sowie *projekt-lokale* Modul-Komponenten. Auf diese drei Sichtbarkeits-Typen wird im Verlauf dieses Dokumentes näher eingegangen (siehe Kapitel 2.9.2 Seite 25).

2.1.1 Validierung von Modulnamen bei der Installation bzw. Aktualisierung

Jedes FSM-Archiv enthält einen Modul-Deskriptor (`/META_INF/module.xml`), der alle Komponenten-Ressourcen, Sichtbarkeits-Stufen, Versionen einzelner Komponenten und Ressourcen sowie deren Abhängigkeiten, Abhängigkeiten zu anderen Modulen oder Modul-Herstellern, Modul- sowie Komponenten-Namen beschreibt (siehe Kapitel 3.8 Seite 39).

Das FSM-Archiv enthält den Namen des Moduls (`<name>`) zur Vergabe eines eindeutigen Modulnamens. Dieser Name wird beispielsweise zur Anzeige in der FirstSpirit Projekt- und Serverkonfiguration, zur Überprüfung des Moduls bei der



Aktualisierung und der Installation und zum Anlegen von Dateien und Ordnern auf der Festplatte verwendet. Der Modulname muss einer bestimmten Namenskonvention genügen (siehe Kapitel 3.8.2.1 Seite 40).

Dabei sind folgende Zeichen erlaubt: A-Za-z0-9; ,_-

Der Name wird beim Installieren des Moduls validiert. Module, die dieser Konvention nicht entsprechen, können nicht installiert werden.

2.1.2 Konfiguration von Modul-Komponenten

Nach erfolgreicher Modul-Installation besteht die Möglichkeit, den ausgewählten Projekten projekt- oder web-lokale Komponenten (siehe Kapitel 2.9.1 Seite 18) über die Projekteigenschaften hinzuzufügen. Optional können diese Komponenten konfiguriert werden (falls vorgesehen und im Modul-Descriptor definiert, `<configurable>`, siehe Kapitel 2.9.3 Seite 26), entweder mit einer von der Komponente erzeugten oder einer generischen GUI. Letztere zeigt die Parameter des Moduls und unterstützt dabei Text und numerische Werte.

2.1.3 Aktivierung von Modul-Komponenten

Web-lokale Komponenten müssen nach ihrer Konfiguration aktiviert werden. In diesem Schritt werden alle Webanwendungen eines Projektes zusammengefasst und abhängig vom verwendeten Webserver deployed. Wird der interne Webserver eingesetzt, ist nichts weiter zu tun. Generische Webserver können das automatische Deployment anbieten, wenn ein entsprechendes Deployment-Skript hierfür existiert. Andernfalls wird eine War-Datei erzeugt, die heruntergeladen und manuell installiert werden muss.

2.2 Aktualisierung eines Moduls

Identisch zur „Installation eines Moduls“ wird bei der Aktualisierung das Modul über die Server- und Projektkonfiguration auf dem FirstSpirit-Server installiert. Die Laufzeitdateien projekt-lokaler Komponenten werden nur dann automatisch aktualisiert, wenn die Komponente(n) beim Hinzufügen in die Projekte entsprechend markiert wurden. Die Aktivierung von Webanwendungen kann ebenfalls über die Server- und Projektkonfiguration angestoßen werden.

Da nach der Aktualisierung eines Moduls die Konfigurationen und Laufzeitdateien projekt-lokaler Komponenten potentiell in einer älteren Version vorliegen können, müssen Module immer mit den Dateien ihrer Vorgängerversionen umgehen können



(lesend und schreibend). Das gilt vor allem für die Konfiguration und die dazugehörige GUI – das Modul muss diesen Anforderungen genügen.

Nach dem Update von Modulen, die Abhängigkeiten zu Modulen mit Diensten ("Service") haben, müssen diese Dienste manuell neugestartet werden.

Bei Installation und Aktualisierung von Modulen, die entweder selbst oder durch direkt oder indirekt abhängige Dienste Grundlage für Daten sind, stehen diese Daten bis zum Neustart der auf diese zugreifenden Prozesse (Generierungen, Clients...) diesen Prozessen nicht mehr zur Verfügung.

Weiterführende Informationen zur Aktualisierung von Modulen über die FirstSpirit-Server- und Projektkonfiguration siehe FirstSpirit Handbuch für Administratoren.

2.3 Deinstallation eines Moduls

Die Modul-Deinstallation wird ebenfalls über die FirstSpirit Server- und Projektkonfiguration durchgeführt. Hierbei werden alle im Modul-Descriptor definierten Komponenten und Ressourcen-Einträge entfernt. Alle dem Modul nicht bekannten bzw. im Modul-Descriptor nicht definierten Ressourcen können beispielsweise über die Definition einer eigens für diese Aufgabe vorgesehenen Modul-Klasse `<class>` (siehe Kapitel 2.6 Seite 14) deinstalliert werden. Im Umkehrschluss gilt dieses auch für die Installation sowie Aktualisierung.

2.4 Import und Export von Modulen

Bei Projektexport-Operationen werden Konfigurations- und Laufzeitdateien *projekt-lokaler* Komponenten ebenfalls exportiert. Während eines Imports besteht die Möglichkeit, ein projekt-lokales Modul dem importierten Projekt hinzuzufügen und die Konfiguration zu übernehmen. Voraussetzung hierfür ist jedoch, dass das Modul auf dem Server installiert ist und in einer gleichen oder einer neueren Version vorliegt.



2.5 Modul-Bestandteile

Im Kern besteht ein Modul aus einem Modul-Descriptor (siehe Kapitel 3.8 Seite 39), Ressourcen und einer bis mehreren Komponenten, wobei jede Komponente wiederum Ressourcen enthalten kann.

2.5.1 Ressourcen

Klassen und andere Ressourcen werden im Modul und in den Komponenten in `<resource>`-Einträgen definiert. Diese verweisen auf eine Jar-Datei oder ein Verzeichnis. Um diese Einträge im Classloader verwenden zu können, werden sie von FirstSpirit ausgepackt (Modul-Installation, Serverstart) und in ein temporäres Verzeichnis gelegt. Daher sollten die Ressourcen möglichst minimal gewählt werden.

```
1. <resources>
2.   <resource>/libs/exmod.jar</resource>
3.   <resource>files/</resource> <!-- Wichtig: "/" am Ende des
4.                               Verzeichnisnamens -->
5.   <resource>libs/simple.jar</resource>
6. </resources>
```

Listing 1: Ressourcen im Module-, Komponenten-Descriptor

Die `<resource>`-Einträge können mit weiteren (optionalen) Attributen zur Versionierung (siehe Kapitel 2.5.1.1 Seite 12) und zum Gültigkeitsbereich (siehe Kapitel 2.5.1.2 Seite 12) versehen werden.

Zur Nutzung namensgleicher Modul-Ressourcen siehe Kapitel 3.19 'Namensgleichheit bei Modul-Ressourcen (z.B. Zip-Exportdateien)' Seite 201.



Alle Ressourcen, einschließlich Zip-Dateien etc., sollten immer mit `MyClass.class.getResourceAsStream(...)` geladen werden.



2.5.1.1 Versionierung

<resource>-Einträge können mit bestimmten Attributen versehen werden, welche einen eindeutigen Bezeichner sowie die mitgelieferte Version (**version**) und die zur aktuellen Version minimal (**minVersion**) und maximal (**maxVersion**) kompatible Versionsnummer definieren.

```
1. <!-- Datei mit Versionsangabe -->
2. <resource name="myLib" version="1.2.7" minVersion="1.2"
   maxVersion="1.2.999" >libs/myLib.jar</resource>
```

Listing 2: Versionierung von Ressourcen im Modul-, Komponenten-Descriptor

Diese Informationen helfen dabei, doppelte Ressourcen bei der Kombination mehrerer Komponenten zu vermeiden. Dies kann beispielsweise bei system-weiten Komponenten oder bei Webanwendungen auftreten.

2.5.1.2 Gültigkeitsbereich

Die Erreichbarkeit bzw. der Gültigkeitsbereich einer Ressource kann über das Attribut **scope** gesteuert werden. Die entsprechenden Jar-Archive, Verzeichnisse und/oder Dateien können entweder nur innerhalb des Moduls (**scope="module"**) oder serverweit (**scope="server"**) erreichbar sein.

```
1. <resource scope="module">libs/mod.jar</resource> <!-- Klassen sind nur
   innerhalb des Moduls erreichbar. -->
2. <resource scope="server">libs/lib.jar</resource> <!-- Klassen sind
   server-weit verfügbar. -->
```

Serverweiter Gültigkeitsbereich: Ist eine Komponente (beispielsweise eine Library-Komponente) serverweit gültig, stehen die enthaltenen Klassen (verpackt in einem oder mehreren Jar-Archiven) nach der Installation auf dem Server, im Client, in Skripten und anderen Modulen ohne weitere Aktivierung zur Verfügung.

Nachteil: Alle Klassen liegen in einen Namespace und damit kann es jede Klasse nur einmal geben. Also: Keine verschiedenen Versionen einer Klasse.

Modul-lokaler Gültigkeitsbereich: Ist eine Komponente (beispielsweise eine Library-Komponente) modul-lokal gültig, so sind die Klassen nur innerhalb des Moduls verfügbar. Jedes Modul sieht "seine" lokalen Bibliotheken und die globalen Bibliotheken. In unterschiedlichen Modulen können damit verschiedene Versionen der gleichen Klasse existieren, ohne sich gegenseitig zu beeinflussen.



Es gilt: Klassen aus global-definierten Jars können *keine* Klassen aus modul-lokal definierten Jars benutzen. Der umgekehrte Weg ist jedoch möglich.

Standardverhalten: Wird kein Gültigkeitsbereich definiert, hängt der Gültigkeitsbereich vom Typ der Komponente ab (siehe Kapitel 2.9.1 Seite 18):

Modul-lokaler Gültigkeitsbereich (standardmäßig, falls nicht anders definiert):

- innerhalb eines Moduls
- alle Komponenten vom Typ ProjectApp
- alle Komponenten vom Typ WebApp
- alle Komponenten vom Typ WebServer

Serverweiter Gültigkeitsbereich (standardmäßig, falls nicht anders definiert):

- alle Komponenten vom Typ Bibliothek
- alle Komponenten vom Typ Editor
- alle Komponenten vom Typ Service
- alle Komponenten vom Typ Public

Hinweis: Um Konflikte zu vermeiden, sollten alle Ressourcen (Jar-Archive), soweit das möglich ist, modul-lokal definiert werden.

2.5.1.3 Komponenten

Ein Beispiel für mehrere in einem FirstSpirit-Modul gekapselte Komponenten ist eine Editor-Komponente (erweitert den FirstSpirit-Client um Eingabemöglichkeiten, ist immer global d.h. serverweit) und eine Service-Komponente (eine Server-Komponente, die eine öffentliche Schnittstelle besitzt und somit über Eingabekomponenten oder Scripte angesprochen werden kann; ist immer global, d.h. serverweit). Ein einfacher Anwendungsfall für ein derartiges Modul bzw. zwei solcher Komponenten wäre z.B. das Einlesen, Bearbeiten, Speichern einer serverseitigen Datei: Einlesen der Datei durch den Service, Bearbeiten durch den Editor in z.B. einer (J)Textarea und anschließendes Speichern derselbigen Datei durch den Editor bzw. durch Auslösen der „Speichern“-Aktion des Editors und anschließender Weitergabe an den Service.



2.6 Modul-Ereignisbehandlung

Auf bestimmte Ereignisse wie Installation, Deinstallation und Aktualisierung durch Benutzer in der Administrationsoberfläche kann durch eine spezialisierte Modul-Klasse `<class>` reagiert werden.

```
<class>de.espirit.firstspirit.module.AnotherClass</class>
```

Des Weiteren implementiert jede Komponente die Methoden `installed()`, `updated()` und `uninstalling()`, siehe auch Kapitel 3.6 Seite 37.

Bestehen Abhängigkeiten zu anderen Modulen, müssen die Modulnamen als `<depends>`-Einträge definiert werden.

```
3.   <dependencies>
4.       <depends>AnotherModule</depends>
5.   </dependencies>
```

2.7 Modul-, Komponenten-Verzeichnisstruktur

Die Verzeichnisstruktur der Module und Komponenten ist an FirstSpirit angepasst, alle Daten-, Konfigurations- und Log-Verzeichnisse sind voneinander getrennt. Nachfolgend ist die Struktur ausgehend vom FirstSpirit Root-Verzeichnis (`$cmscroot`) dargestellt:



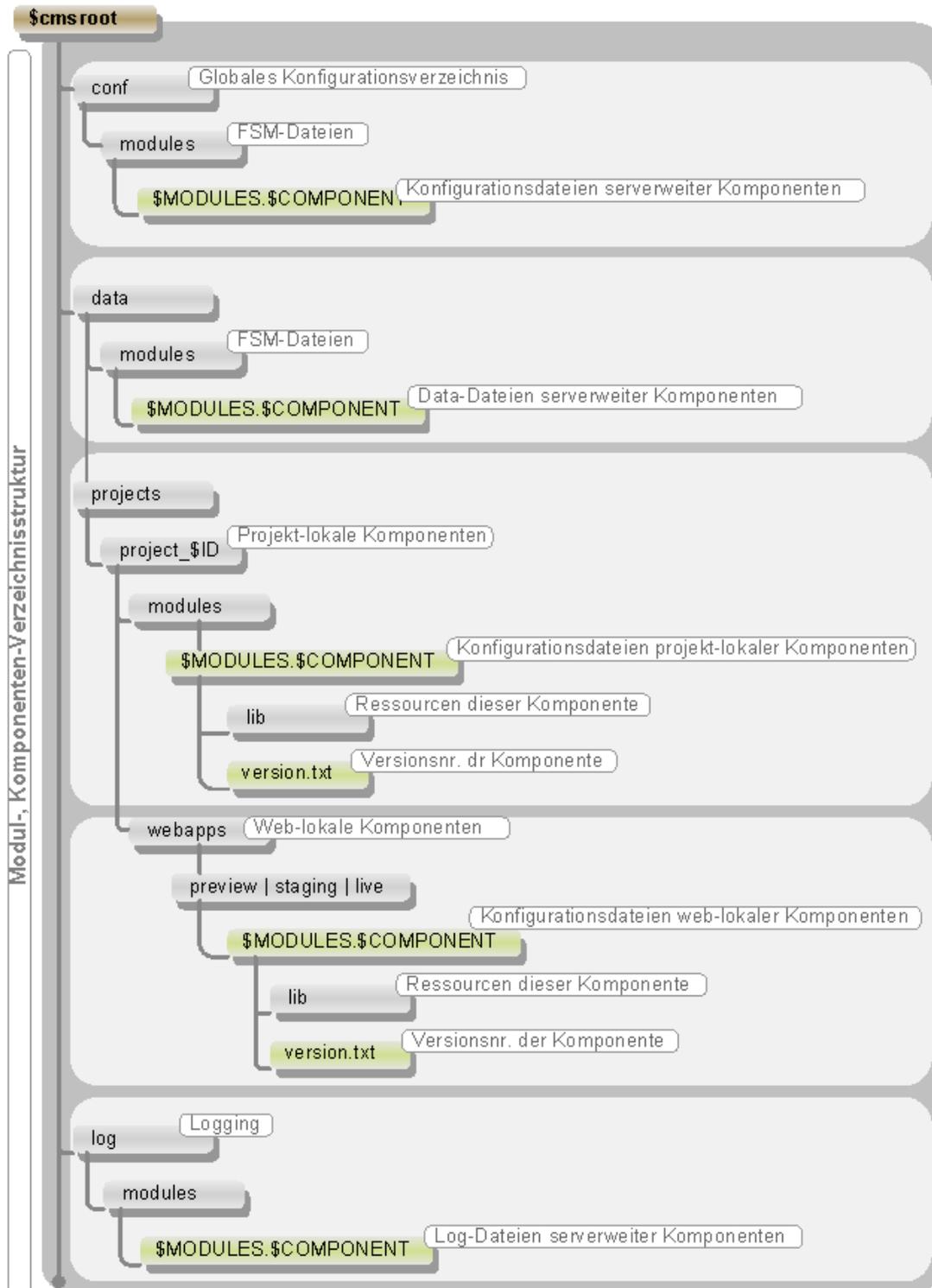


Abbildung 2-1: Module/Komponenten Verzeichnisstruktur





`$MODULES . $COMPONENT` steht für `Modulname.Komponentenname`

```
1.
2.     <module>
3.         <name>MyModule</name>
4.         <components>
5.             <service>
6.                 <name>MyService</name>
7.             </service>
8.         </component>
9.     </module>
10.
```



Aus Zeile 3 und Zeile 6 ergibt sich dann der Verzeichnisname:
`MyModule.MyService`



2.8 FSM-Archiv-Struktur

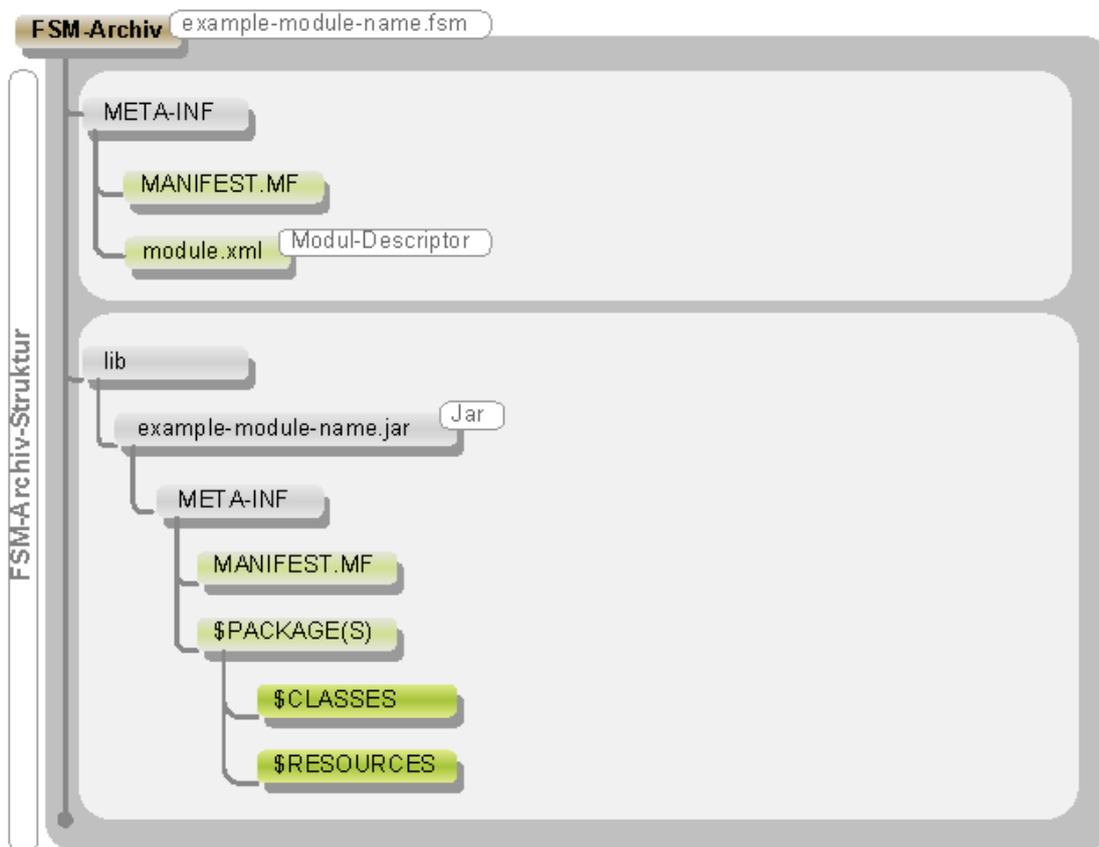


Abbildung 2-2: FSM-Archiv-Verzeichnisstruktur



2.9 Komponenten

Komponenten erweitern den `<components>`-Teil des Modul-Descriptors (siehe Kapitel 3.8 und 3.8.1 ab Seite 39).

```
1. <components>
2.     <-- (siehe 3.8) -->
3. </components>
```

2.9.1 Komponenten-Typen

2.9.1.1 Bibliothek

Eine Bibliothek ist die einfachste Form einer Komponente. Sie ist eine (meist konfigurationslose) Sammlung von Klassen, verpackt in einem oder mehreren Jar-Archiven. Bibliotheken implementiert das Interface `Library`:

- `de.espirit.firstspirit.module.Library`

Bibliotheken erweitern den `<components>`-Teil des Modul-Descriptors (Beispiel siehe Kapitel 3.9.1.1 Seite 41):

```
<library></library>
```

Sichtbarkeit: Bibliotheken sind immer server-weit sichtbar, d.h. sie stehen nach der Installation auf dem Server, im Client, in Scripten und anderen Modulen ohne weitere Aktivierung zur Verfügung (siehe Kapitel 2.9.2 Seite 25).

Die Definition einer Library-Komponente im Modul-Deskriptor kann über Ressourcen innerhalb des Library-Elements (s.o.) oder als Modul-Ressource umgesetzt werden (vgl. „Gültigkeitsbereich“ in Kapitel 2.5.1.2). Vorteil der zweiten Variante, es können Archive mit gleichnamigen Klassen in unterschiedlichen Versionen vorliegen und parallel zur Verwendung kommen (vgl. „Komponentenlose Modul-Implementierung (JDBC-Connector-Modul)“ in Kapitel 3.15 Seite 149).

Konfiguration: Eine Library-Komponente kann eine Konfigurationsoberfläche zur Verfügung stellen, die innerhalb der FirstSpirit Server- und Projektkonfiguration aufgerufen werden kann. Die entsprechende Konfigurationsklasse muss dazu das Interface `Configuration<ServerEnvironment>` implementieren (siehe Kapitel 2.9.3 Seite 26).



2.9.1.2 Editoren (veraltet)



Dieses Kapitel bezieht sich ausschließlich auf die Gui-Editoren (Interface `Editor<E extends ServerEnvironment>`), die mit FirstSpirit Version 5 auslaufen. Die Entwicklung neuer Eingabekomponenten wird ab FirstSpirit 42R4 auf `SwingGadgets` umgestellt (siehe Kapitel 3.2.1 Seite 34). `SwingGadgets` werden als Public-Komponente (siehe Kapitel 3.9.1.6 Seite 47 bzw. Kapitel 2.9.1.7 Seite 23) im Descriptor eingebunden.

Eine Editor-Komponente stellt eine Eingabekomponente inklusive Datenmodell- und Render-Klasse zur Verfügung, über die es möglich ist, den Client um eigene Eingabemöglichkeiten zu erweitern. In der Regel werden FirstSpirit-Eingabekomponenten entwickelt, wenn dem Redakteur entweder eine neue Funktionalität zur Verfügung gestellt oder eine projektspezifische Aufgabe vereinfacht werden soll.

`CMS_INPUT_PERMISSION` ist ein Beispiel für einen eigenen Editor, der die Definition von Rechten erlaubt. Er arbeitet mit einem passenden Service zusammen, der die Gruppendefinitionen vom Server lädt und bereitstellt.

Editoren implementiert das typisierte Interface `Editor<ServerEnvironment>`:

- `de.espirit.firstspirit.module.Editor`
- `de.espirit.firstspirit.module.ServerEnvironment`
- `de.espirit.firstspirit.client.access.editor.swing.AbstractValueGuiEditor`.

Editoren erweitern den `<components>`-Teil des Modul-Descriptors (Beispiel siehe Kapitel 3.9.1.2 Seite 42):

```
<editor></editor>
```

2.9.1.3 Projektanwendung

Eine Projektanwendung ergänzt FirstSpirit-Projekte um bestimmte Fähigkeiten bzw. Eigenschaften. Projektanwendungen implementiert das Interface `ProjectApp`:

- `de.espirit.firstspirit.module.ProjectApp`

Projektanwendungen erweitern den `<components>`-Teil des Modul-Descriptors (Beispiel siehe Kapitel 3.9.1.7 Seite 50):



```
<project-app></project-app>
```

Sichtbarkeit: Projektanwendungen sind projekt-lokal sichtbar, d.h. sie müssen nach der Installation des Moduls den gewünschten Projekten hinzugefügt werden (siehe Kapitel 2.9.2 Seite 25).

Konfiguration: Eine Projektanwendung kann eine Konfigurationsoberfläche zur Verfügung stellen, die innerhalb der FirstSpirit Server- und Projektkonfiguration aufgerufen werden kann (Projekteigenschaften – Projektkomponenten). Die entsprechende Konfigurationsklasse muss dazu das typisierte Interface `Configuration<ProjectEnvironment>` implementieren (siehe Kapitel 2.9.3 Seite 26):

- `de.espirit.firstspirit.module.Configuration`
- `de.espirit.firstspirit.module.ProjectEnvironment`

2.9.1.4 Service

Ein Service ist eine Server-Komponente, ausgestattet mit einem öffentlichen Interface. Über den `ServiceLocator` der FirstSpirit-Access-API [2] ist der Service serverweit verfügbar. Über die öffentliche Schnittstelle können Eingabekomponenten (Editoren) oder Scripte den Dienst ansprechen. Als Beispiel ist hier eine Virenscan-Modul-Implementierung zu nennen, die im Verlauf dieses Dokumentes näher erläutert wird (siehe Kapitel 3.17 Seite 155).

Services implementiert das Interface `Service`:

- `de.espirit.firstspirit.module.Service`
- `de.espirit.firstspirit.access.ServiceLocator`

Services erweitern den `<components>`-Teil des Modul-Descriptors (Beispiel siehe Kapitel 3.9.1.3 Seite 44):

```
<service></service>
```

Sichtbarkeit: Services sind immer server-weit sichtbar, d.h. sie stehen nach der Installation auf dem Server, im Client, in Scripten und anderen Modulen ohne weitere Aktivierung zur Verfügung (siehe Kapitel 2.9.2 Seite 25).

Konfiguration: Eine Service-Komponente kann eine Konfigurationsoberfläche zur Verfügung stellen, die innerhalb der FirstSpirit Server- und Projektkonfiguration aufgerufen werden kann (Servereigenschaften - Module). Die entsprechende Konfigurationsklasse muss dazu das Interface `Configuration<ServerEnvironment>` implementieren (siehe Kapitel 2.9.3



Seite 26):

- `de.espirit.firstspirit.module.Configuration`
- `de.espirit.firstspirit.module.ServerEnvironment`
- `de.espirit.firstspirit.module.ServiceProxy`

2.9.1.5 Webanwendung

```
<web-app></web-app>
```

Eine Webanwendung dient der Definition von JSP-Tags und Servlets, die anschließend in den Projekten verwendet und aufgerufen werden können. Aus allen für ein Projekt definierten Webanwendungen muss bei der Aktivierung (Deployment) eine einzige Webanwendung mit einer gemeinsamen `web.xml` entstehen, deren XML-Einträge einer fest vorgegebenen Reihenfolge genügen müssen. Hierzu werden die folgenden Elemente aus den einzelnen `web.xml`-Templates herausgelesen und in die gemeinsame `web.xml` kopiert:

```
<context-param>,  
  <filter>,  
  <filter-mapping>,  
  <listener>,  
  <servlet>,  
  <servlet-mapping>,  
  <mime-mapping>,  
  <error-page>,  
  <taglib>.
```

Alle übrigen Einträge werden ignoriert. Context-, Filter- und Servlet-Namen müssen eindeutig sein. Bei der Entwicklung muss dies berücksichtigt und mit passendem Präfix gewährleistet werden. Gleiches gilt für Taglib-URLs. Konflikte bei Mime-Mappings und Fehlerseiten dagegen werden ignoriert und nach dem Prinzip `first-come-first-serve` aufgelöst. Konfigurationsdateien, die von der Komponente selbst geschrieben wurden, werden im `WEB-INF`-Verzeichnis der Webanwendung abgelegt und stehen den Servlets über die Methode

`ServletContext.getResourceAsStream()` zur Verfügung. Stehen im `web.xml`-Template Platzhalter in der Form `${propertyKey}`, so werden diese mit den Werten aus der Konfiguration ersetzt.

Webanwendungen implementiert das Interface `WebApp`:

- `de.espirit.firstspirit.module.WebApp`



Webanwendungen erweitern den `<components>`-Teil des Modul-Descriptors (Beispiel siehe Kapitel 3.9.1.8 Seite 51):

```
<web-app></web-app>
```

Sichtbarkeit: Webanwendungen sind web-lokal sichtbar, d.h. sie müssen nach der Installation des Moduls den gewünschten Webbereichen im Projekten hinzugefügt werden (siehe Kapitel 2.9.2 Seite 25 und Kapitel 2.1.3 Seite 9).

Konfiguration: Eine Webanwendung kann eine Konfigurationsoberfläche zur Verfügung stellen, die innerhalb der FirstSpirit Server- und Projektkonfiguration aufgerufen werden kann (Projekteigenschaften – Webkomponenten). Die entsprechende Konfigurationsklasse muss dazu das typisierte Interface `Configuration<WebEnvironment>` implementieren (siehe Kapitel 2.9.3 Seite 26):

- `de.espirit.firstspirit.module.Configuration`
- `de.espirit.firstspirit.module.WebEnvironment`

2.9.1.6 Webserver

Eine Webserver-Komponente steuert einen laufenden Webserver. Beispiele hierfür sind die interne Webserver-Steuerung oder eine Tomcat-Unterstützung. Eine Webserver-Komponente implementiert die Steuerung eines bestimmten Webserver (beispielsweise Tomcat oder Jetty), zunächst das Deploy/Undeploy von Webanwendungen. Über die FirstSpirit Server- und Projektkonfiguration (Servereigenschaften - Webserver) lassen sich Webserver hinzufügen und so dem FirstSpirit-Server bekannt machen.

Im System vorhandene Webserver:

- **Intern:** Steuerung des internen Jetty.
- **Generic:** Deploy/Undeploy über Beanshell-Skripte, die mit einer einfachen Key-Value-Map konfiguriert werden können. Die Skripte liegen im Konfigurationsverzeichnis (Kapitel 2.7 Seite 14) und lassen sich über den Konfigurations-Dialog editieren.
- **Extern:** Leere Implementierung; unterstützt kein Deploy oder Undeploy.

Anwendungsbeispiel:

- Die Unterstützung für WebSphere wird als Modul (`fs-ibmws.fsm`) umgesetzt.
- Installation des Moduls.
- Server-Eigenschaften: WebSphere als "WS-1" hinzufügen, URLs und Passwörter



konfigurieren.

- Projekt-Eigenschaften: Staging, Umschalten von "Intern" auf "WS-1".

Webserver implementiert das Interface `WebServer`:

- `de.espirit.firstspirit.module.WebServer`
- `de.espirit.firstspirit.access.schedule.WebServerConfiguration`
- `de.espirit.firstspirit.access.schedule.DeployTarget`

Webserver-Komponenten erweitern den `<components>`-Teil des Modul-Deskriptors (Beispiel siehe Kapitel 3.9.1.4 Seite 45):

```
<web-server></web-server>
```

Sichtbarkeit: Webserver-Komponenten sind immer server-weit sichtbar, d.h. sie stehen nach der Installation auf dem Server zur Verfügung. Nach der Installation kann der Webserver den gewünschten Webbereichen (Preview, Staging, Live) eines (oder mehrerer) Projekte zugeordnet werden (Projekteigenschaften – Webkomponenten) (siehe Kapitel 2.9.2 Seite 25).

Konfiguration: Eine Webserver-Komponente kann eine Konfigurationsoberfläche zur Verfügung stellen, um beispielsweise Pfade, URLs, Passwörter u.ä. einzustellen, die innerhalb der FirstSpirit Server- und Projektkonfiguration aufgerufen werden kann (Servereigenschaften - Webserver). Die entsprechende Konfigurationsklasse muss dazu das Interface `Configuration<ServerEnvironment>` implementieren (siehe Kapitel 2.9.3 Seite 26):

- `de.espirit.firstspirit.module.Configuration`
- `de.espirit.firstspirit.module.ServerEnvironment`

2.9.1.7 Public

Neben komplexeren Komponenten wie Editoren oder Webanwendungen existieren im FirstSpirit-System auch Schnittstellen/HotSpots, für die kein spezialisierter Komponenten-Typ definiert wurde. Über die Public-Deklaration des Moduls werden Klassen dem FirstSpirit-Server bekannt gemacht, die solche Schnittstellen implementieren. Diese Klassen müssen über Modul-Bibliotheken (Library-Komponente, siehe Kapitel 2.9.1.1 und 2.5.1 Seiten 18 bzw. 11) oder Modul-Ressourcen analog zu einer komponentenlosen Library (siehe Kapitel 3.15 ‚Komponentenlose Modul-Implementierung (JDBC-Connector-Modul)‘ Seite 149 und Absatz 2.9.1.1 ‚Bibliothek‘, Seite 18) gefunden werden.

- Bei diesem Typ ist keine Implementierung (`implements`) eines speziellen Interfaces notwendig.



Public-Komponenten erweitern den `<components>`-Teil des Modul-Descriptors (Beispiel siehe Kapitel 3.9.1.5 Seite 46):

```
<public></public>
```

Sichtbarkeit: Public-Komponenten sind immer server-weit sichtbar, d.h. sie stehen nach der Installation auf dem Server, im Client, in Scripten und anderen Modulen ohne weitere Aktivierung zur Verfügung (siehe Kapitel 2.9.2 Seite 25).

Die Definition einer Public-Komponente im Modul-Deskriptor kann über Ressourcen innerhalb des Public-Elements (s.o.) oder als Modul-Ressource umgesetzt werden (vgl. „Gültigkeitsbereich“ in Kapitel 2.5.1.2).

Konfiguration: Eine Public-Komponente kann eine serverweite Konfigurationsoberfläche zur Verfügung stellen, die innerhalb der FirstSpirit Server- und Projektkonfiguration aufgerufen werden kann (Servereigenschaften – Module). Die entsprechende Konfigurationsklasse muss dazu das Interface `Configuration<ServerEnvironment>` implementieren (siehe auch Kapitel 2.9.3 Seite 26).

Ausführbare Klassen: Ressourcen, die als Public-Komponente im Deskriptor eingebunden werden, können ausführbare Implementierungen enthalten. Nach der Installation des Moduls stehen diese ausführbaren Implementierungen auf dem FirstSpirit-Server zur Verfügung und können beispielsweise innerhalb eines Skripts (Auftrags-Skripte im FirstSpirit-Server oder Skripte im FirstSpirit-JavaClient) aufgerufen werden. Um einen globalen Zugriff auf diese Dateien zu gewährleisten, musste für diese Ressourcen bisher (bis FirstSpirit Version 4.2R4/Build 4.2.434) ein serverweiter Gültigkeitsbereich definiert werden. Das ist nicht in allen Fällen erwünscht. Da beispielsweise Klassen aus global-definierten Jars *keine* Klassen aus modul-lokal definierten Jars verwenden können, mussten auch alle weiteren Ressourcen mit serverweitem Gültigkeitsbereich definiert werden (vgl. Gültigkeitsbereich von Ressourcen in Kapitel 2.5.1.2).

Ab FirstSpirit Version 4.2R4 (Build 4.2.434) kann eine ausführbare Implementierung auch dann referenziert werden, wenn die Klasse in einem modul-lokalen Jar-Archiv liegt. Um Konflikte zu vermeiden, sollten alle Ressourcen (Jar-Archive), soweit das möglich ist, modul-lokal definiert werden.



Beispiel (Ressourcen werden mit modul-lokalem Gültigkeitsbereich definiert):

```
<module>
  <name>Modul 1</name>
  <version>1.0</version>
  <components>
    <public>
      <name>AppExecutor-Name</name>
      <class>de.espirit.pm.modules.modul.AppExecutor</class>
    </public>
  </components>
  <resources>
    <resource scope='module'>lib/modul1.jar</resource>
    <resource scope='module'>lib/jtwitter_api.jar</resource>
  </resources>
</module>
```

Referenzierung der ausführbaren Klasse im Skript-Source-Tab:

```
<snip>
#!executable-class
AppExecutor-Name
  //this calls class de.espirit.pm.modules.modul.AppExecutor registered in
  //"Module 1"
</snip>
```

Ab FirstSpirit Version 42R4 wird die Implementierung von Eingabekomponenten auf SwingGadgets umgestellt (siehe Kapitel 3.2.1 Seite 34). Diese werden ebenfalls als Public-Komponente (siehe Kapitel 3.9.1.6 Seite 47) im Deskriptor eingebunden.

2.9.2 Sichtbarkeit von Komponenten

Jede Komponente hat innerhalb der FirstSpirit-Umgebung eine bestimmte Sichtbarkeit. Dabei sind einige Komponenten generell serverweit sichtbar, hierzu gehören folgende Komponenten: Editor, Webserver, Library (Bibliothek), Service und Public (siehe auch Kapitel 3.9.1 Seite 41).

Server: Serverweite Komponenten stehen nach ihrer Installation auf dem kompletten Server zur Verfügung – damit auch im JavaClient, in allen Scripten sowie in allen anderen Komponenten.

Projekt: Projekt-lokale Komponenten werden manuell an einzelne Projekte gebunden. Konfigurations- und Laufzeitdateien werden bei diesem Komponententyp im Projektverzeichnis abgelegt.

Web: Web-lokale Komponenten (zu denen per Definition die Webanwendungen gehören) werden manuell den einzelnen Web-Bereichen Preview, Staging und Live bestimmter Projekte zugefügt. Das ermöglicht nicht nur verschiedene Konfigurationen, sondern es können auch Komponenten nur für bestimmte Bereiche aktiviert werden.



2.9.3 Konfiguration von Komponenten

Jede Komponente innerhalb der FirstSpirit-Umgebung kann konfiguriert werden. Dazu muss zunächst eine Klasse für die grafische Konfigurationsoberfläche erstellt werden, die das typisierte Interface `Configuration<E extends ServerEnvironment>` implementiert:

```
public class MyComponentConfigPanel implements
Configuration<ServerEnvironment> {
    ...
}
```

Das Interface ist typisiert, d. h. der zu verwaltende Wertetyp wird über die (Java 5) Generics-Funktionalität innerhalb der Implementierung festgelegt. Abhängig vom Komponententyp (vgl. Kapitel 2.9.1) bzw. vom gewünschten Gültigkeitsbereich (vgl. Kapitel 2.5.1.2) wird hier eine mehr oder weniger umfangreiche Schnittstelle verwendet, die die Umgebung eines installierten Moduls widerspiegelt:

- Interface `ServerEnvironment`: Wird für serverweit-gültige Komponenten (beispielsweise Bibliotheken) verwendet und stellt nützliche Informationen bereit, wie z.B. das Modul-Konfigurations-Verzeichnis oder das Modul-Verzeichnis für Logdateien (siehe Kapitel 2.11 Seite 31).
- Interface `ProjectEnvironment` Wird für projektlokal-gültige Komponenten (Projektanwendungen – siehe Kapitel 2.9.1.3) verwendet und erweitert die Informationen aus dem Interface `ServerEnvironment`, um projektspezifische Informationen, wie z.B. die Project-ID.
- Interface `WebEnvironment`: Wird für weblokal-gültige Komponenten (Webanwendungen – siehe Kapitel 2.9.1.5) verwendet und erweitert die Informationen aus dem Interface `ProjectEnvironment`, um webspezifische Informationen, wie den Zugriff auf die `web.xml` der betreffenden Webanwendung.



Anschließend muss innerhalb des Komponenten-Deskriptors das Attribut `<configurable/>` hinzugefügt werden:

```
<module>
  ...
  <components>
    ...
    <public>
      <configurable> MyComponentConfigPanel </configurable>
    </public>
  </components>
</module>
```

Nach der Installation des Moduls ist die entsprechende Komponente konfigurierbar. Abhängig vom Komponententyp bzw. vom Gültigkeitsbereich erfolgt diese Konfiguration serverweit, projekt- oder weblokal. Der Button „Konfigurieren“ in der FirstSpirit Server- und Projektkonfiguration (Servereigenschaften – Module bzw. Projekteigenschaften – Projekt- bzw. Web-Komponenten) ist aktiv.

Beispiel siehe Virensan-Modul-Implementierung in Kapitel 3.17 Seite 155.



2.10 Classloading

2.10.1 Hierarchie

Um Konflikte zwischen verschiedenen Modulen und dessen Komponenten zu vermeiden, werden die Klassen – soweit wie möglich – über einzelne, voneinander getrennte Classloader geladen. Hierbei gilt die folgende Classloader-Hierarchie:

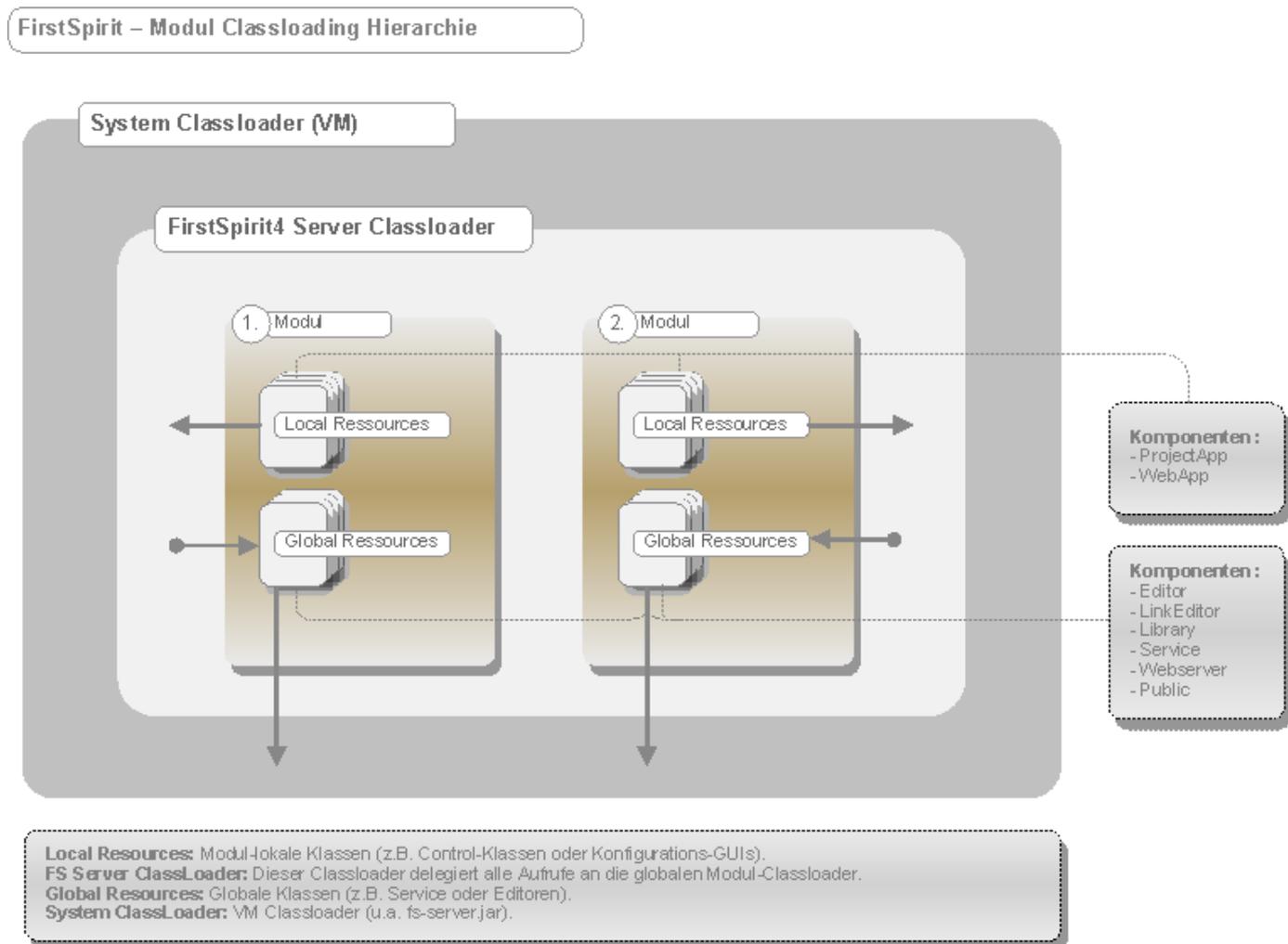


Abbildung 2-3: Modul-Classloading-Hierarchie

Die nachfolgende Grafik zeigt die Modul-Ressourcen-Aufteilung in die einzelnen Classloader. Das Beispiel-Modul in der folgenden Grafik enthält eine Projekt- und eine Web-Komponente (lokal), einen Editor (global) sowie eine Library und einen Service (global). Außerdem sind noch Modul-Ressourcen für die Modul-Steuerungs-Klasse definiert.



FirstSpirit – Aufteilung der Module-Ressourcen in die einzelnen Classloader

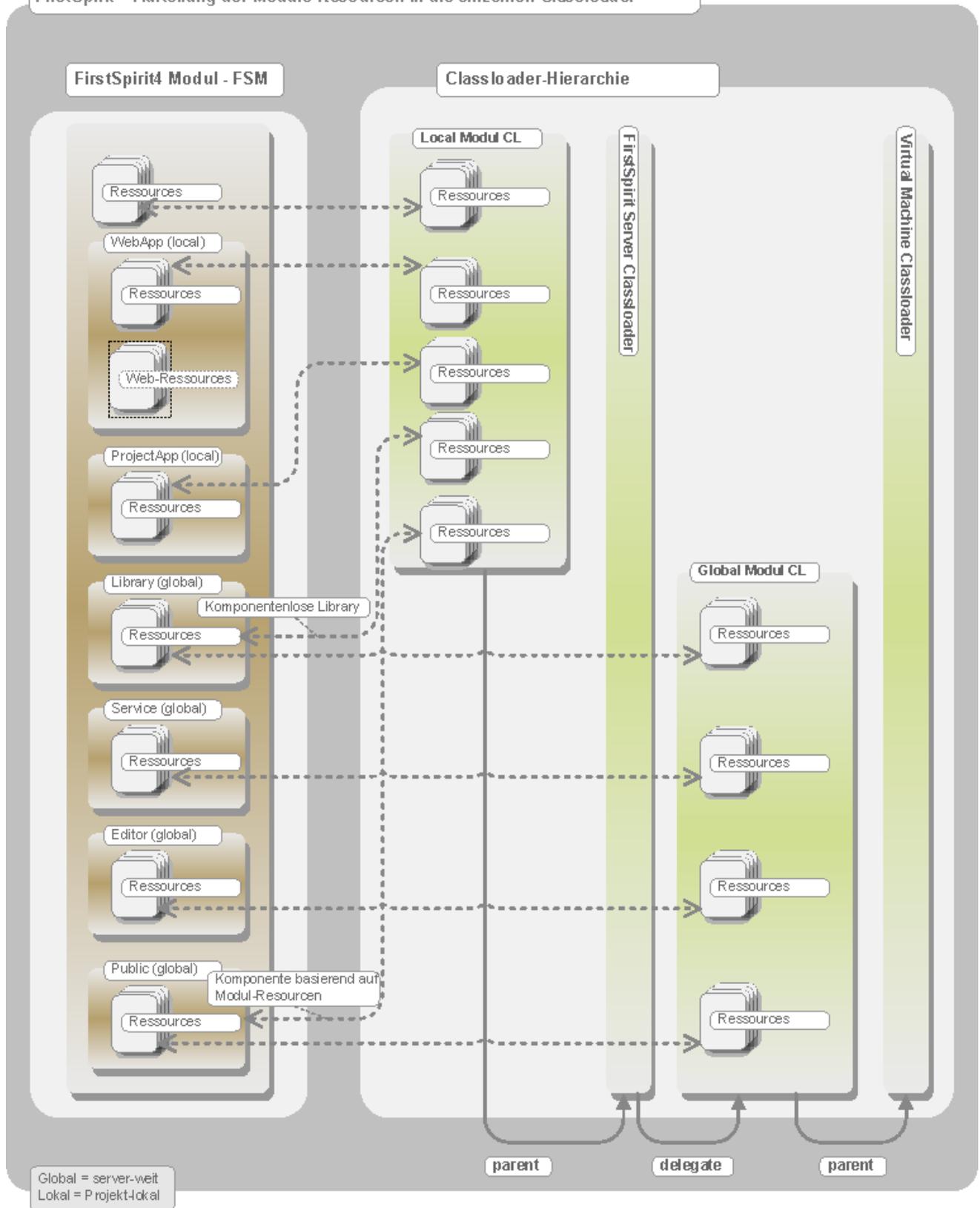


Abbildung 2-4: Modul-Classloading-Hierarchie – Classloader-Aufteilung



Alle lokalen Komponenten werden zusammen mit den Modul-Ressourcen im Local Module-Classloader zusammengefasst.

Die serverweiten Ressourcen der Library und des Services gelangen in den Global Module-Classloader. Dieser ist nicht direkt, sondern nur über den FirstSpirit-Server-Classloader erreichbar. Dieser delegiert alle `findClass()`- und `findResource()`-Aufrufe an die Module weiter. Die Classloading-Hierarchie ermöglicht, dass die lokalen Klassen (wie die Konfigurations-GUI) globale Klassen (beispielsweise einen Service) verwenden können.

2.10.2 Classloading in Webanwendungen am Beispiel FirstSpirit u. WebSphere

Anwendungsfall:

Eine Webanwendung, die in WebSphere 6.1 eine FirstSpirit-Connection hält, und über diese Verbindung einen Modul-Service mittels `getService(„ServiceName“)` remote vom FirstSpirit-Server lädt, wird unter Umständen eine `ClassCastException` werfen, da die Klasse des Services, die vom FirstSpirit-Server zurückgeliefert wird, von zwei verschiedenen Classloadern geladen wird. Die Klasse des Service, die zurückgeliefert wird, ist vom FirstSpirit-RemoteClassloader geladen worden und kann nicht auf die Klasse gecastet werden, die der Anwendung durch den WebSphere-Classloader (CompoundClassloader) bekannt ist.

Die Classloader-Hierarchie stellt sich in etwa wie folgt dar:

```
WebApp (CompoundClassloader) → VM-Classloader
```

Im VM-Classloader liegt das `fs-webrt.jar`, unter `WEB-INF` liegt der Service und somit im WebApp-Classloader, dieser delegiert an VM-Classloader. Da die FirstSpirit-Connection (VM-Classloader) versucht, die Service-Klasse zu laden und nichts findet, lädt sie diese über den FirstSpirit-RemoteClassloader. Dieser wird in der Hierarchie zwischen WebApp und VM nicht erreicht und daher wird die Service-Klasse von zwei Classloadern geladen.

Lösung:

Das JAR der Modul-Service-Komponente und das FirstSpirit `fs-webrt.jar` müssen in diesem Fall an gleicher Stelle liegen, also entweder im VM-Classloader oder im WebApp-Classloader (`WEB-INF/lib`).



2.11 Komponenten-Konfigurationsdateien

Hier muss zwischen globalen (serverweiten) Komponenten und lokalen (projekt-lokalen) Komponenten unterschieden werden. Konfigurationsdateien globaler Modul-Komponenten befinden sich im Verzeichnis:

```
conf/module/$MODULE.$COMPONENT,
```

Konfigurationsdateien lokaler Komponenten sind im Verzeichnis:

```
data/projects/project_$ID/conf
```

 zu finden bzw. sollten hier abgelegt werden.

```
FileHandle handle = _environment.getConfDir().obtain(„my-  
module.conf“);
```

Das ServerEnvironment **_environment** wird beim Aufruf bzw. bei der Initialisierung der Komponente durch die FirstSpirit Modul-Umgebung über die `init()`-Methode des jeweiligen Komponenten-Interfaces übergeben (siehe Kapitel 3.4 Seite 36). Das ServerEnvironment (**de.espirit.firstspirit.module.ServerEnvironment**) stellt nützliche Informationen bereit, wie z.B. das Modul-Konfigurations-Verzeichnis oder das Modul-Verzeichnis für Logdateien. Die Methode `getConfDir()` liefert das Konfigurationsdateien-Verzeichnis für das jeweilige Environment. Die Komponente sollte alle Konfigurationsdateien in diesem Verzeichnis ablegen.

Beispiel `init()`-Methode einer Service-Komponente:

```
1. public void init(final ServiceDescriptor descriptor, final  
   ServerEnvironment env) {  
2.     _environment = env;  
3. }
```



2.12 Logging in FirstSpirit

2.12.1 Globales Logging – fs-server.log / fs-client.log

```
4. import de.espirit.common.base.Logging;
5.
6. private static final Class<?> LOGGER = MyClassName.class;
7.
1. Logging.logInfo("...the string to log", LOGGER);
2. Logging.logWarn("...the string to log", LOGGER);
3. Logging.logDebug("...the string to log", LOGGER);
4. Logging.logTrace("...the string to log", LOGGER);
5. Logging.logError("...the string to log", LOGGER);
```

Listing 3: Globales Logging

2.12.2 Lokales Logging auf Modulebene

Um auf Modulebene eine Logdatei in `log/module/$MODULE.$COMPONENT` zu schreiben (ein einfaches Anwendungsbeispiel für eine Logdatei wäre die Installation oder Aktualisierung), siehe auch Kapitel 2.7 Seite 14, kann über das `ServerEnvironment` (siehe Kapitel 2.11 Seite 31) das hierfür in der globalen Verzeichnisstruktur vorgesehene Logverzeichnis bezogen werden. Die Methode `getLogDir()` liefert das Logdateien-Verzeichnis für das jeweilige Environment. Die Komponente sollte alle Logdateien in diesem Verzeichnis speichern. Für diese Art des Loggings wird eine modulspezifische Implementierung benötigt z.B. Log4J oder SLF4J.

```
FileHandle handle = _environment.getLogDir().obtain(„my-  
module.log“);
```

Listing 4: Logging auf Modulebene



3 Das FirstSpirit 5 Modul- / Komponenten-Modell



Mit FirstSpirit-Version 5.0 wird ein neues, flexibleres Modul- und Komponenten-Modell eingeführt. Ab FirstSpirit Version 4.2R4 sollte diese neue Infrastruktur verwendet werden. Einige der in diesem Kapitel beschriebenen Vorgehensweisen, Klassen und Methoden, insbesondere im Bereich der Implementierung von Eingabekomponenten, können damit veraltet oder unvollständig sein. Die Überarbeitung erfolgt baldmöglichst.

3.1 Übersicht

Alle Komponenten-Typen in FirstSpirit lassen sich untereinander kombinieren, so dass ein komponenten-basierter Informationsfluss möglich ist. Unter bestimmten Anforderungen ist diese Kombination auch zwingend erforderlich.



Soll beispielsweise eine Editor-Komponente auf einen entfernten Web-Service zugreifen, wird dieses durch die clientseitige FirstSpirit-Security-Policy verhindert. Die Implementierung eines solchen Zugriffs muss über einen serverseitigen Service umgesetzt werden, d.h. die Eingabekomponente (Editor-Komponente) kommuniziert mit dem Service über das FirstSpirit-Protokoll bzw. über das ServerEnvironment besteht die Möglichkeit, Verbindungsinformationen zu beziehen und so über die aktuelle Verbindung (`getConnection()`) mit dem spezialisierten Service zu kommunizieren. Der Service stellt die benötigte Verbindung mit dem entfernten Web-Service her und delegiert alle Daten weiter an den Web-Service bzw. zurück an die Editoren-Komponente.

```
// a) Get the service implementation by class
Service =
    _environment.getConnection().getService(MyService.class);

// b) Get the service implementation by descriptor name
MyService service = (MyService)
    _environment.getConnection().getService("MyService");
```

Listing 5: ServerEnvironment – Service



- a) Die erste Variante nutzt hier die konkrete Klasse und sollte aus Gründen der Typensicherheit bevorzugt genutzt werden.
- b) Der String „MyService“ entspricht hier dem definierten Namen im Modul-Deskriptor.

```
<name>MyService</name>
```

`_environment.getConnection()` – bietet außerdem grundlegende Methoden zur Beschaffung von serverseitigen Daten, beispielsweise

```
_environment.getConnection().getProjects()
_environment.getConnection().getProjectById(long id)
_environment.getConnection().getProjectByName(String name)
```

Listing 6, zu finden in der FirstSpirit-Access-API

3.2 Restrukturierung mit FirstSpirit 5

Mit FirstSpirit-Version 5.0 wird ein neues, flexibleres Modul- und Komponenten-Modell eingeführt.

3.2.1 Änderungen zur FirstSpirit Version 4

Die wesentlichen Änderungen zum Komponenten-Modell der FirstSpirit-Version 4, beziehen sich auf die Eingabekomponenten des FirstSpirit-JavaClients, die so genannten SwingGadgets. Ein SwingGadget ist die grafische Repräsentation einer Eingabekomponente in der FirstSpirit-Redaktionsumgebung (JavaClient) – das Pendant zu den Gui-Editoren der Version 4. Jedes SwingGadget (Editor) kann funktionale Aspekte implementieren. Funktionale Aspekte für SwingGadgets sind beispielsweise ValueHolder, Singlelineable, Labelable oder Editable. Wobei ValueHolder, Labelable u. Editable keine optionalen Aspekte sind, sondern in der abstrakten Basis-Implementierung jedem SwingGadget hinzugefügt werden. Der Modul-Deskriptor und die darin enthaltenen Komponenten-Deskriptoren (siehe Kapitel 3.8 Seite 39) ändern sich hingegen nur minimal.



SwingGadgets der Version 5 sind kompatibel zur FirstSpirit-Version 4.2R4.



3.2.2 Zielsetzung: Einfach, effizient und erweiterbar

Dank neuer Verfahren, sogenannte „Aspekte“, müssen Kernfunktionalitäten im neuen FirstSpirit-Komponentenmodell nicht von der SwingGadget-Implementierung selber umgesetzt werden, sondern stehen als Interface (Aspekt-Typen) zur Verfügung. Die SwingGadget-Implementierung muss nur noch die entsprechenden Methoden des Interfaces implementieren, das umliegende FirstSpirit-Gadget-Framework behandelt dann automatisch alle weiteren Funktionen, beispielsweise das Speichern eines Wertes (siehe Kapitel 3.12.2 Seite 69). Ein weiterer Vorteil der Aspekte ist die Möglichkeit zur kompatiblen Erweiterbarkeit der Eingabekomponenten. Soll eine Komponente mit einer neuen Funktionalität versehen werden, kann das einfach über einen neuen Aspekt realisiert werden. Die bisherige Implementierung bleibt kompatibel und muss nicht angepasst werden.

Außerdem wird durch die Restrukturierung eine klare Trennung von Visualisierung (GUI) und Datenträgerherkunft angestrebt. Dadurch können komplexe Eingabekomponenten wie beispielsweise FS_LIST mit projektspezifischen Datenträgern (über Services) verwendet werden. Somit muss bei der Entwicklung neuer Eingabekomponenten bzw. für neue Einsatzzwecke nur eine Datenträger-Schnittstelle (in der Regel ein Dienst) implementiert werden, aber keine neue GUI, da hier auf bereits vorhandene Eingabekomponenten zurückgegriffen werden kann.

3.3 Komponenten-Container (Typen)

Wie schon zuvor in Kapitel 2.9.1 (ab Seite 18) erwähnt, gibt es sechs Komponenten-Typen oder Komponenten-Container.

- Eine **Bibliothek** implementiert das typisierte Interface `Library<ServerEnvironment>`
- eine **Editor**-Komponente implementiert das typisierte Interface `Editor<ServerEnvironment>`
(Die Unterstützung für dieses Interface läuft in FirstSpirit Version 5 aus – die Implementierung neuer Eingabekomponente sollte auf SwingGadgets umgestellt werden.)
- eine **Projektanwendung** implementiert das Interface `ProjectApp`, die optional eine Konfigurationsoberfläche der Projektanwendung implementiert, ggf. das Interface `Configuration<ProjectEnvironment>`;
- ein **Service** implementiert das Interface `Service<T>` (implementierende Klassen müssen einen no-arg-Konstruktor besitzen), die eine mögliche Service-Konfigurationsmaske implementiert `Configuration<ServerEnvironment>`;



- eine **Webanwendung** erweitert `AbstractWebApp` und implementiert das Interface `WebServer`;
- eine **Webserver**-Komponente implementiert das Interface `Webserver`.

```
de.espirit.firstspirit.module.Library
de.espirit.firstspirit.module.Editor
de.espirit.firstspirit.module.ServerEnvironment
de.espirit.firstspirit.module.ProjectApp
de.espirit.firstspirit.module.Configuration
de.espirit.firstspirit.module.WebEnvironment
de.espirit.firstspirit.module.ProjectEnvironment
de.espirit.firstspirit.module.ServerEnvironment
de.espirit.firstspirit.module.Service
de.espirit.firstspirit.access.ServiceLocator
de.espirit.firstspirit.module.ServiceProxy
de.espirit.firstspirit.module.WebApp
de.espirit.firstspirit.module.WebServer
```

3.4 Initialisierung von Komponenten

Die Komponenten werden vom FirstSpirit-System automatisch initialisiert. Der Zeitpunkt der Initialisierung einer Komponente ist abhängig vom Komponenten-Typ.

- **Editoren** werden „on demand“ geladen, d.h. bei der ersten Benutzung der jeweiligen Eingabekomponente im JavaClient
- **Projektanwendungen** werden bei der Projektinitialisierung geladen
- **Bibliotheken**, **Services** (wenn die Einstellung Autostart aktiv ist), **Webanwendungen**, **Webserver**-Komponenten werden beim Serverstart geladen.

Eine Komponente und ggf. vorhandene Konfigurationsoberflächen werden durch den Aufruf folgender Methoden initialisiert (Die Methoden müssen durch jede Komponenten implementiert werden) (siehe Kapitel 2.11 Seite 31):

1. Aufruf des parameterlosen Konstruktors der Komponente
2. Die `init`¹-Methode wird aufgerufen.

Beispiele zur Initialisierung finden sich in:

¹ `de.espirit.firstspirit.module.Component#init(D extends ComponentDescriptor, E extends ServerEnvironment)`



- Kapitel 3.17.5 (Beispiel Service).
- Kapitel 3.17.7 (Beispiel Library).
- Kapitel 3.18.2.2 (Beispiel WebApp).

Sind mehrere Komponenten in einem Modul-Deskriptor (Kapitel 3.8 Seite 39) definiert, werden alle Komponenten durch den Aufruf ihrer jeweiligen `init`¹-Methoden initialisiert.

3.5 Entladen von Komponenten

Das Entladen von Komponenten ist unter FirstSpirit 4.2R4 ebenfalls vereinfacht worden. Beim Aktualisieren eines Moduls können laufende Instanzen des FirstSpirit JavaClients nun im laufenden Betrieb über die aktuellen Änderungen informiert werden. Im Einzelfall kann jedoch ein Refresh im JavaClient (beispielsweise bei einer Editor-Komponente) oder auch ein Neustart des JavaClients notwendig sein.

3.6 Event-Methoden von Komponenten

Jede Komponente (mit Ausnahme der SwingGadgets), die ihr spezifisches Interface implementiert, muss auf bestimmte Ereignisse wie Installation, Deinstallation und Aktualisierung durch Benutzer in der FirstSpirit Server- und Projektkonfiguration reagieren (vgl. Kapitel 2.6 Seite 14) und verwendet dazu die folgenden Event-Methoden:

- `public void init(D descriptor, E env)`: Initialisierung der Komponente und ggf. vorhandener Konfigurationsoberflächen (siehe auch Kapitel 3.4 Seite 36).
- `public void installed()`: Diese Methode wird aufgerufen, nachdem die Komponente bzw. das Modul erfolgreich auf dem FirstSpirit-Server installiert wurde.
- `public void uninstalling()`: Diese Methode wird aufgerufen, wenn die Komponente deinstalliert wird.
- `public void updated(final String oldVersionString)`: Diese Methode wird aufgerufen, wenn die Komponente auf dem FirstSpirit-Server aktualisiert wird.

```
1.  /**
2.   * Initializes this component with the given {@link
   * de.espirit.firstspirit.module.descriptor.ComponentDescriptor
3.   * descriptor} and {@link
```



```
de.espirit.firstspirit.module.ServerEnvironment environment}. No other
method will be called
4.     * before the component is initialized!
5.     *
6.     * @param descriptor useful descriptor information for this component.
7.     * @param env         useful environment information for this component.
8.     */
9.     public void init(final ServiceDescriptor descriptor, final
ServerEnvironment env) {
10.         _environment = env;
11.     }
12.     -----
13.     /** Event method: called if Component was successfully installed (not
updated!) */
14.     public void installed() {
15.         copyConfigFile(getConfigFileName()); // NON-NLS
16.         Logging.logDebug(getName() + " installed...", LOGGER); // NON-
NLS
17.         appendLog("installed"); // NON-NLS
18.     }
19.     -----
20.     /** Event method: called if Component was in uninstalling procedure
*/
21.     public void uninstalling() {
22.         Logging.logDebug(getName() + " uninstalling...", LOGGER);
23.         // NON-NLS
24.     }
25.     -----
26.
27.     /**
28.     * Event method: called if Component was completely updated
29.     *
30.     * @param oldVersionString old version, before component was updated
31.     */
32.     public void updated(final String oldVersionString) {
33.         // e.g. merge, diff, bkup the config file or do not overwrite
34.         copyConfigFile(getConfigFileName()); // NON-NLS
35.         Logging.logDebug(getName() + " updated...", LOGGER); // NON-NLS
36.         appendLog("updated"); // NON-NLS
37.     }
38.
```

Listing 7: Komponenten Event-Methoden – installed, updated, uninstalled



Beispiel siehe:

- Implementierung einer Service-Komponente (siehe Kapitel 3.17.5 Seite 160).
- Implementierung einer Webanwendung (siehe Kapitel 3.18.2.1 Seite 191).

3.7 Modul Datei-, Archiv-Format (.fsm)

Alle Bestandteile eines Moduls werden in einem FSM(Zip)-Archiv mit dem Suffix `.fsm` zusammengefasst (siehe Kapitel 3.20 Seite 203). Signierte Module werden bei der Installation überprüft und gegebenenfalls zurückgewiesen (siehe Kapitel 3.21 Seite 203).

3.8 Der Modul-Deskriptor

Das FSM-Archiv muss einen Modul-Deskriptor enthalten (/META-INF/module.xml), der Auskunft über das Modul und die darin gekapselten Komponenten gibt.

3.8.1 Beispiel Modul-Deskriptor

```
1. <!DOCTYPE module SYSTEM "http://www.FirstSpirit.de/module.dtd">
2. <module>
3.   <name>ExampleModule</name>
4.   <version>1.0</version>
5.   <description>This module is just an example.</description>
6.   <vendor>e-Spirit AG</vendor>
7.   <class>de.espirit.exmod.ModuleImpl</class>
8.   <resources>
9.     <resource>libs/exmod.jar</resource>
10.  </resources>
11.  <components>
12.    <!-- (,Komponenten-Deskriptor' siehe unter 0 2.9.1 u. 03.9) -->
13.  </components>
14.  <dependencies>
15.    <depends>AnotherModule</depends>
16.  </dependencies>
17. </module>
```

Listing 8: Modul-Deskriptor-Beispiel



3.8.2 Modul-Deskriptor Tags und Attribute

3.8.2.1 Pflichtfelder

Folgende Felder sind für jeden Modul-Deskriptor zwingend erforderlich:

```

1. <module>
2.     <name></name>
3.     <version></version>
4.     <components></components>
5. </module>
    
```

<module> <u>Mandatory</u>	Der einleitende Modul-Deskriptor-Container
<name> <u>Mandatory</u>	Name des Moduls. Erlaubte Zeichen: A-Za-z0-9; ,_- Der Name wird beim Installieren des Moduls validiert. Module, die dieser Konvention nicht entsprechen, können nicht installiert werden (siehe Kapitel 2.1.1 Seite 8).
<version> <u>Mandatory</u>	Modulversionierung Punkt-Notation z.B. 0.1 oder 0.0.1 oder 0.0.0.1 + optionale Revision z.B. _72
<components> <u>Mandatory</u>	Hier werden die im Modul enthaltenen Komponenten definiert (siehe auch Kapitel 3.9 Seite 41)

Listing 9: Pflichtfelder des Modul-Deskriptors

3.8.2.2 Optionale Einträge

```

1. <description></description>
2. <vendor></vendor>
3. <class></class>
4. <resources>
5.     <resource></resource>
6. </resources>
    
```

<description>	Aussagekräftige Beschreibung des Moduls und dessen Funktionalität
<vendor>	Eine Herstellerangabe, z.B. e-Spirit AG



<class>	Auf bestimmte Ereignisse wie Installation, Deinstallation und Aktualisierung durch Benutzer in der Administrationsoberfläche kann durch eine spezialisierte Modul-Klasse reagiert werden, um bspw. Aktionen an allen Komponenten eines Moduls auszuführen.
<resources>	Container für 0..* Ressourcen-Einträge
<resource>	Der Ressourcen-Eintrag
<dependencies>	Abhängigkeiten-Container, kann 0..* abhängige Moduleinträge enthalten
<depends>	Besitzt das Modul Abhängigkeiten zu anderen Modulen, müssen die Modulnamen an dieser Stelle als <depends>-Einträge definiert werden.

Listing 10: Optionale Modul-Deskriptor-Elemente

3.9 Der Komponenten-<components>-Deskriptor-Teil

Komponenten erweitern den <components>-Teil des Modul-Deskriptors (siehe Kapitel 3.8.1 Seite 39). Das <components>-Element ist zwingend erforderlich auch wenn es sich um eine komponentenloses Modul handelt (siehe Kapitel 3.15 „Komponentenlose Modul-Implementierung (JDBC-Connector-Modul)“, Seite 149).

```

1.    ...
2.    <components>
3.    <!-- <library> <editor> <project-app> <web-app> <web-server> -->>>>
4.    </components>
5.    ...

```

3.9.1 Komponenten-Deskriptoren und spezielle Eigenschaften

Der Komponenten-Deskriptor ist ein „Teil“-Deskriptor des Modul-Deskriptors.

3.9.1.1 Bibliothek

Eine **Bibliothek** ist die einfachste Form einer Komponente. Sie ist eine (meist konfigurationslose) Sammlung von Klassen, verpackt in einem oder mehreren Jar-Archiven. Serverweite Bibliotheken stehen nach der Installation auf dem FirstSpirit-Server, im JavaClient, in Skripten und anderen Modulen ohne weitere Aktivierung zur Verfügung. Projekt- und weblokale Bibliotheken müssen zunächst einem Projekt hinzugefügt werden.



```

1. <library>
2.   <name>MyLibs</name>
3.   <description>Example library.</description>
4.   <hidden>true</hidden>
5.   <resources>
6.     <!-- shared resources -->
7.     <resource name="log4j" version="1.2.7" minVersion="1.2"
maxVersion="1.2.999">libs/log4j.jar</resource>
8.     <resource name="JAMon" version="1.0">libs/JAMon.jar</resource>
9.     <resource name="poi" version="2.5">libs/poi.jar</resource>
10.  </resources>
11. </library>

```

Eigenschaften	
<hidden>	Optional. Definiert ob eine Komponente „in der Modul-Konfiguration der Server- und Projekteigenschaften“ zu sehen ist. false true [true]
<name> <i>Mandatory</i>	Der Name der Komponente ist in einer Bibliothek zwingend erforderlich, dies gilt für alle Komponenten.
<resources> <i>Mandatory</i>	Das <i><resources></i> -Element muss definiert sein, auch wenn dieses keine weiteren Ressourcen-Einträge enthält.
<configurable>	Optional. Definition der Konfigurationsoberflächen-Klasse. Ist dieses Element nicht definiert, stellt die Komponente in der FirstSpirit Server- und Projektkonfiguration keine Konfigurations-GUI zur Verfügung – der „Konfigurieren“-Button in der FirstSpirit Server- und Projektkonfiguration ist deaktiviert.

Listing 11: Library Komponenten-Deskriptor und Eigenschaften

3.9.1.2 Editor (veraltet)



Dieses Kapitel bezieht sich ausschließlich auf die Gui-Editoren (Interface Editor<E extends ServerEnvironment>), die mit FirstSpirit Version 5 auslaufen. Die Entwicklung neuer Eingabekomponenten wird ab FirstSpirit 42R4 auf SwingGadgets umgestellt (siehe Kapitel 3.2.1 Seite 34). SwingGadgets werden als Public-Komponente (siehe Kapitel 3.9.1.6 Seite 47) im Deskriptor eingebunden.

Eine **Editor**-Komponente stellt eine Eingabekomponente inklusive Datenmodell- und Render-Klasse (`<class> Attribute`) zur Verfügung, über die es möglich ist, den



Client um eigene Eingabemöglichkeiten zu erweitern.

```

1.   <editor>
2.     <name>CMS_INPUT_SIMPLE_EDITOR_CONTENT</name>
3.     <description>FirstSpirit Simple Example Editor</description>
4.     <class>
de.espirit.firstspirit.opt.example.editor.simple.FSEditorContentComponent
     </class>
5.     <resources>
6.       <resource version="0.0.1" minVersion="0.0"
maxVersion="0.0.1">lib/fs-seditor-example.jar</resource>
7.     </resources>
8.   </editor>
    
```

Eigenschaften:

<name> <u>Mandatory</u>	Definiert den Namen, über den die Komponente in FirstSpirit als Eingabekomponente eingebunden wird.
<class> <u>Mandatory</u>	Klasse, die das typisierte Interface <code>de.espirit.firstspirit.module.Service<T></code> implementiert. konkret: <code>VScanServiceImpl</code> implements <code>VScanService</code> , <code>de.espirit.firstspirit.module.Service<VScanService></code>
<resource> <u>Mandatory</u>	Die Editor-eigenen Ressourcen. version Aktuelle Version (optional) minVersion Minimal kompatible Version (optional) maxVersion Maximal kompatible Version (optional)
<configurable>	Optional. Definition der Konfigurationsoberflächen-Klasse. Ist dieses Element nicht definiert, stellt die Komponente in der FirstSpirit Server- und Projektkonfiguration keine Konfigurations-GUI zur Verfügung – der „Konfigurieren“-Button in der FirstSpirit Server- und Projektkonfiguration ist deaktiviert.

Listing 12: Editor Komponenten-Deskriptor und Eigenschaften



3.9.1.3 Service

Ein **Service** ist eine Server-Komponente, ausgestattet mit einem öffentlichen Interface. Über den `de.espirit.firstspirit.access.ServiceLocator` der FirstSpirit Access-API [2] ist der Service serverweit verfügbar. Über die öffentliche Schnittstelle können z.B. Eingabekomponenten (SwingGadgets) oder Skripte den Dienst ansprechen. Als Beispiel ist hier die Virenskan-Modul-Implementierung zu nennen, welchen im Verlauf dieses Dokumentes näher erläutert wird (siehe Kapitel 3.17).

```

1.   <service>
2.       <name>VScanService</name>
3.       <description>FirstSpirit Virus Scan Service</description>
4.       <class>de.espirit.firstspirit.opt.vscan.VScanServiceImpl</class>
5.       <configurable>de.espirit.firstspirit.opt.vscan.admin.gui.VScanServiceC
   onfigPanel</configurable>
6.       <resources>
7.           <resource name="libvscan">lib/fs-vscan.jar</resource>
8.           <resource>fs-vscan.conf</resource>
9.       </resources>
10.  </service>

```

Eigenschaften:

<name> <u>Mandatory</u>	Definiert den Namen, über den die Komponente in FirstSpirit als Service-Komponente erreichbar ist.
<class> <u>Mandatory</u>	Klasse, die das typisierte Interface <code>de.espirit.firstspirit.module.Service<T></code> implementiert. konkret: <i>VScanServiceImpl implements VScanService, de.espirit.firstspirit.module.Service<VScanService></i>
<configurable>	Optional. Definition der Konfigurationsoberflächen-Klasse. Ist dieses Element nicht definiert, stellt der Service in der FirstSpirit Server- und Projektkonfiguration keine Konfigurations-GUI zur Verfügung – der „Konfigurieren“-Button in der FirstSpirit Server- und Projektkonfiguration ist deaktiviert. konkret: <i>VScanServiceConfigPanel implements de.espirit.firstspirit.module.Configuration<ServerEnvironment></i>



<resource> <u>Mandatory</u>	<p>Definition der Service-eigenen Ressourcen. Jeder Service muss zwingend seine "eigenen" Klassen als Ressourcen in Form eines Jars enthalten.</p> <ul style="list-style-type: none"> ▪ lib/fs-vscan.jar ▪ fs-vscan.conf (Service-Properties-Datei wird bei der Installation des Moduls in das vorgesehene Verzeichnis kopiert, siehe Kapitel 2.7 und 2.8 ab Seite 14) <p>version Aktuelle Version (optional) minVersion Minimal kompatible Version (optional) maxVersion Maximal kompatible Version (optional)</p>
---	---

Listing 13: Service Komponenten-Deskriptor und Eigenschaften

Eine Beispiel-Service-Implementierung befindet sich im Zip-Archiv zum Modul-Entwicklerhandbuch (MODDEV4x_modexamples.zip).



Jeder Service ist out-of-the-box Multithreaded, d.h. ein ExecutorService oder andere nebenläufige Implementierungen innerhalb der Service-Implementierung sind nicht nötig. Jeder Remote-Request an den Service wird indirekt über den FirstSpirit-NIOSocketServer angesprochen, somit ist die Nutzung/Anfrage des Services, z.B. aus einer Eingabe-Komponente oder wie im Beispiel der Virens Scanner-Modul-Implementierung automatisch Multithreaded. Alle Anfragen/Threads werden vom FirstSpirit-NIOSocketServer verwaltet.

3.9.1.4 Webserver

Die Webserver-Komponente dient zurzeit nur zur Installation bzw. De-Installation von Webanwendungen.



3.9.1.5 Public

Eine **Public**-Komponente (Schnittstelle/HotSpot) ist eine spezialisierte Klasse, die eine Schnittstelle der FirstSpirit-Access-API implementiert, für die kein spezialisierter Komponenten-Typ definiert wurde. Über die Public-Anweisungen des Moduls werden dem FirstSpirit-Server Klassen bekannt gemacht, die solche Schnittstellen implementieren. Diese Klassen müssen über Modul-Bibliotheken (siehe Kapitel 2.9.1.1 und 2.5.1 Seite 18 bzw. 11) gefunden werden. Die Definition der Bibliothek kann auch innerhalb des gleichen Moduls erfolgen, welches die Public-Komponente definiert.

```

1.   <public>
2.     <name>Beanshell</name>
3.     <description>Beanshell Scripting engine.</description>
4.
   <class>de.espirit.firstspirit.server.script.BeachellScriptEngine</class>
5.   </public>

```

Listing 14: Public Komponenten-Deskriptor und Eigenschaften

Eigenschaften:	
<name> <i>Mandatory</i>	Definiert den Namen, über den die Komponente in FirstSpirit als Public-Komponente erreichbar ist.
<class> <i>Mandatory</i>	Klasse, die ein Interface der FirstSpirit-Access-API implementiert.
<configurable>	Optional. Definition der Konfigurationsoberflächen-Klasse. Ist dieses Element nicht definiert, stellt die Komponente in der FirstSpirit Server- und Projektkonfiguration keine Konfigurations-GUI zur Verfügung – der „Konfigurieren“-Button in der FirstSpirit Server- und Projektkonfiguration ist deaktiviert.

Wobei hier die Klasse `BeachellScriptEngine` das Interface `ScriptEngine` implementiert. Eine mögliche Kombination einer Service- und einer Public-Komponente definiert sich im Modul-Deskriptor wie in Kapitel 3.17.4 (Seite 158) zu sehen.



Public-Komponenten benötigen zwingend einen parameterlosen Konstruktor.



3.9.1.6 Gadget-Spezifikation

Eine **SwingGadget**-Implementierung stellt eine Eingabekomponente (SwingGadget) inklusive GOM-Implementierung und Factory-Klassen zur Verfügung, über die es möglich ist, den Client um eigene Eingabemöglichkeiten zu erweitern. SwingGadgets werden als Public-Komponente im Deskriptor eingebunden, erhalten über die Gadget-Spezifikation (siehe <class>) jedoch erweiterte Konfigurationsmöglichkeiten:

```

1.   <public>
2.     <name>CUSTOM_TEXTAREA</name>
3.     <description>FirstSpirit Swing Gadget Editor .</description>
4.     <class>de.espirit.firstspirit.module.GadgetSpecification</class>
5.     <configuration>
6.       <gom>de.espirit.firstspirit.opt.examples.gadgettextarea.GomCustomTextarea</gom>
7.       <factory>de.espirit.firstspirit.opt.examples.gadgettextarea.CustomTextareaSwingGadgetFactory</factory>
8.       <value>de.espirit.firstspirit.opt.examples.gadgettextarea.TextareaValueFactory</value>
9.       <scope data="yes" content="yes" link="yes"/>
10.    </configuration>
11.    <resources>
12.      <resource>lib/gadget-textarea-example-@VERSION@.jar</resource>
13.    </resources>
14.  </public>

```

Eigenschaften:

<name> Mandatory	Definiert den Namen, über den die Komponente in FirstSpirit als Public-Komponente erreichbar ist.
<class> Mandatory	Klasse, die für die Konfiguration des SwingGadgets verantwortlich ist. Die Angaben unter <configuration> bilden die Grundlage für die Konfiguration dieser Klasse.



<p><configuration> <u>Mandatory</u></p>	<p>Die Struktur bzw. die Darstellung eines SwingGadgets wird bestimmt durch die zugehörige GOM-Implementierung (Xml-Beschreibung) und die zugehörigen Factory-Klassen (Instanziert eine neue grafische Repräsentation des Editors). Diese werden innerhalb der <configuration>-Tags angegeben:</p> <ul style="list-style-type: none"> ▪ <gom></gom> Klasse, die für die Xml-Beschreibung der Eingabekomponente benötigt wird. ▪ <factory></factory> Factory-Klasse, die für die Instanzierung einer neuen grafischen Darstellung des SwingGadgets benötigt wird. ▪ Ist der Editor zusätzlich ein ValueHolder muss außerdem in ▪ <value></value> die Factory-Klasse angegeben werden, die für die Behandlung der Werte eines SwingGadgets verantwortlich ist. <p>Innerhalb der <configuration>-Tags kann außerdem der Gültigkeitsbereich der Eingabekomponente innerhalb des FirstSpirit-JavaClients definiert werden.</p> <ul style="list-style-type: none"> ▪ <scope/> Über die Attribute data, content, link und unrestricted kann definiert werden, in welchem Vorlagenbereich eine Eingabekomponente (SwingGadget) verwendet werden darf: <ul style="list-style-type: none"> ○ data="yes": Eingabekomponente kann innerhalb einer Seiten- und/oder Absatzvorlage verwendet werden. ○ content="yes": Eingabekomponente kann innerhalb einer Tabellenvorlage verwendet werden. ○ link="yes": Eingabekomponente kann innerhalb einer Verweisvorlage verwendet werden. ○ unrestricted="yes": Es gibt keine Einschränkungen bei der Verwendung der Eingabekomponente.
<p><resource> <u>Mandatory</u></p>	<p>Die SwingGadget-eigenen Ressourcen. Jede SwingGadget-Komponente muss zwingend seine "eigenen" Klassen als Ressourcen in Form eines Jars enthalten.</p> <p>version Aktuelle Version (optional) minVersion Minimal compatible Version (optional) maxVersion Maximal compatible Version (optional)</p>



3.9.1.6.1 Public-Classes (Schnittstellen für eine Implementierung)

- **Interface *UploadFilter.class*** – für eine UploadFilter-Implementierung als PUBLIC-Komponente, (siehe auch Modul-Implementierung mit den Komponenten-Typen – PUBLIC, SERVICE, LIBRARY, Seite 155), steht bereits eine abstrakte Umsetzung zu Verfügung (***FileBasedUploadFilter.class***), welche auch in dem zuvor referenzierten Beispiel im Detail beschrieben wird.
- **Interface *GomIncludeValueProvider.class*** – HotSpot-Implementierung für das FirstSpirit Gui-Object-Model (GOM-Form). Eine konkrete Umsetzung kann beliebige Mengenwertige Listen aufnehmen, aus welchen der Benutzer dann z.B. eine *CMS_INPUT_COMBOBOX*-Komponente selektieren kann. Eine solche Mengenwertige Liste kann über die GOM-Form-XML-Defintion in die Komponente eingebunden werden.

```
1. <CMS_INCLUDE_OPTIONS type="public">
2.   <NAME>VALUE</NAME>
3.   <PARAMS>
4.     <PARAM name="Classname" />
5.   </PARAMS>
6. </CMS_INCLUDE_OPTIONS>
```



Konkrete Beispiele finden sich innerhalb der FirstSpirit Modul-Developer-API (dev-api) im Zip-Archiv (MODDEV4x_modexamples.zip).



3.9.1.7 Projekt-Anwendung

Eine **Projekt-Anwendung** ist projekt-lokal, d.h. sie muss nach der Installation des Moduls zunächst den gewünschten Projekten hinzugefügt werden. Die selektierten Projekte werden mit bestimmten Eigenschaften/Fähigkeiten ausgestattet. Über eine Konfigurationsoberfläche kann dieser Komponenten-Typ für das jeweilige Projekt konfiguriert werden. Eine Projekt-Anwendung implementiert das Interface `ProjectApp`, eine Konfigurationsmaske für die Projekt-Anwendung implementiert `Configuration<ProjectEnvironment>`.

```

1.   <project-app>
2.     <name>MyProjectConfiguration</name>
3.     <description>My Project Configuration to do some.</description>
4.     <class>
    de.espirit.firstspirit.opt.myprojectapp.MyProjectApp</class>
5.     <configurable>
    de.espirit.firstspirit.opt.myprojectapp.MyProjectConfigurationGui</configurable>
6.   </project-app>
    
```

Eigenschaften:	
<name> <u>Mandatory</u>	Definiert den Namen, über den die Komponente in FirstSpirit als Projekt-Komponente erreichbar ist und in der Liste der zugehörigen Komponenten des Moduls erscheint.
<class> <u>Mandatory</u>	Klasse, die das Interface <code>de.espirit.firstspirit.module.ProjectApp</code> implementiert. konkret: <i>MyProjectApp implements de.espirit.firstspirit.module.ProjectApp</i>
<configurable>	Optional. Definition der Konfigurationsoberflächen-Klasse. Ist dieses Element nicht definiert, stellt die Projekt-Anwendung in der FirstSpirit Server- und Projektkonfiguration kein Konfiguration-GUI zur Verfügung – der „Konfigurieren“-Button in der FirstSpirit Projektkonfiguration ist deaktiviert. Die Konfigurationsklasse muss das typisierte Interface <code>de.espirit.firstspirit.module.Configuration<ProjectEnvironment></code> implementieren. konkret: <i>MyProjectConfigurationGui implements de.espirit.firstspirit.module.Configuration<ProjectEnvironment></i>



3.9.1.8 Webanwendung

Eine **Webanwendung** definiert JSP-Tags und Servlets, die innerhalb eines Projekts verwendet und aufgerufen werden können (siehe Kapitel 2.9.1.5 Seite 21). Eine Webanwendung leitet sich von `AbstractWebApp` ab. Eine Konfigurationsmaske für die Webanwendung kann über die Implementierung von `Configuration<WebEnvironment>` erfolgen.

```

1. <web-app>
2.     <name>My WebApp</name>
3.     <description>Longer description of the webapp</description>
4.     <class>de.espirit.firstspirit.opt.examples.WebApp</class>
5.     <configurable>
6.         de.espirit.firstspirit.opt.examples.WebAppConfiguration
7.     </configurable>
8.     <web-xml>web.xml</web-xml>
9.     <resources>
10.        <resource>lib/webapp-example.jar</resource>
11.    </resources>
12.    <web-resources>
13.        <resource>HelloWorld.tld</resource>
14.        <resource>configuration.properties</resource>
15.        <resource>lib/webapp-example-webapp.jar</resource>
16.    </web-resources>
17. </web-app>

```

Eigenschaften:	
<name> <i>Mandatory</i>	Definiert den Namen, über den die Komponente in FirstSpirit als Projekt-Komponente erreichbar ist und in der Liste der zugehörigen Komponenten des Moduls erscheint.
<description>	Aussagekräftige Beschreibung des Moduls und dessen Funktionalität
<class> <i>Mandatory</i>	Klasse, die die abstrakte Klasse <code>de.espirit.firstspirit.module.AbstractWebApp</code> implementiert. konkret: <i>WebApp extends AbstractWebApp</i>
<web-xml>	Pfad zur web.xml innerhalb der Modulstruktur.



<configurable>	<p>Definition der Konfigurationsoberflächen-Klasse. Ist dieses Element nicht definiert, stellt die Webanwendung in der FirstSpirit Server- und Projektkonfiguration kein Konfiguration-GUI zur Verfügung – der „Konfigurieren“-Button in der FirstSpirit Projektkonfiguration ist deaktiviert. Die Konfigurationsklasse muss das typisierte Interface <code>de.espirit.firstspirit.module.Configuration<WebEnvironment></code> implementieren.</p> <p>konkret: <i>WebAppConfiguration implements Configuration<WebEnvironment></i></p>
<resource> <i>Mandatory</i>	<p>Definition der eigenen Ressourcen. Jede Webanwendung muss zwingend seine „eigenen“ Klassen als Ressourcen in Form eines Jars enthalten.</p> <ul style="list-style-type: none"> ▪ lib/webapp-example.jar <p>version Aktuelle Version (optional) minVersion Minimal kompatible Version (optional) maxVersion Maximal kompatible Version (optional)</p>
<web-resources>	<p>Enthält alle Ressourcen, die auf dem Webserver zur Verfügung stehen müssen. Diese werden in die vom FirstSpirit-Server erzeugte War-Datei übernommen.</p> <ul style="list-style-type: none"> ▪ lib/webapp-example-webapp.jar ▪ HelloWorld.tld ▪ configuration.properties



3.10 Getting started – Erste Schritte zur Modulentwicklung

3.10.1 FirstSpirit Version 4.2R4

1. Das fs-client.jar muss im Classpath liegen.
2. Die Access-API Dokumentation sollte in die IDE integriert werden.
Das Zip-Archiv (MODDEV4x_modexamples.zip) enthält einige konkrete Implementierungs-Beispiele im Verzeichnis `examples`, sowie die FirstSpirit Developer-API Dokumentation, welche nicht zu verwechseln mit der FirstSpirit Access-API Dokumentation ist. Die Developer-API Dokumentation ist speziell zur Modulentwicklung vorgesehen und befindet sich im Verzeichnis `javadoc` als Zip-Archiv.
3. Zum Kompilieren der Implementierungs-Beispiele aus dem Zip-Archiv (MODDEV4x_modexamples.zip) müssen die entsprechenden jars ebenfalls im Classpath liegen. Für die Beispiel-WebApp-Implementierung (siehe Kapitel 3.18 Seite 186) werden das `servlet-api.jar` und `jsp-api.jar` benötigt. Aus lizentechnischen Gründen werden die Beispiel-Implementierungen ohne diese Libraries ausgeliefert. Diese müssen vom jeweiligen Distributor heruntergeladen und im Classpath abgelegt werden.



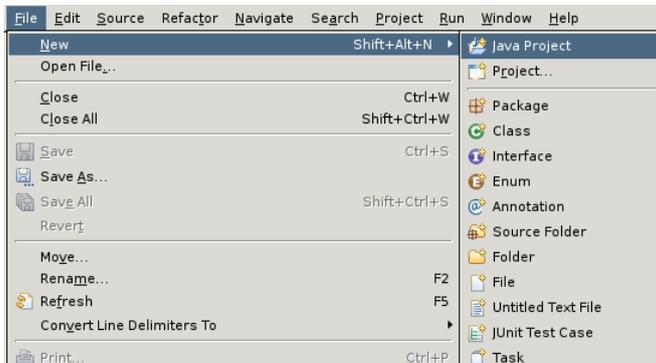
Die Beispiele, die eine JDBC-Modul-Library verwenden, werden aus lizentechnischen Gründen ohne die entsprechende JDBC-Bibliothek ausgeliefert. Diese muss vom jeweiligen Distributor heruntergeladen und im lib-Verzeichnis des Moduls abgelegt werden. Dazu muss die txt-Datei, die als Platzhalter dient, durch die entsprechende Bibliothek ersetzt werden, z.B. „mysql-connector-java-3.1.14-bin.jar.txt“ im Verzeichnis Library/MySQL03/lib durch „mysql-connector-java-3.1.14-bin.jar.“



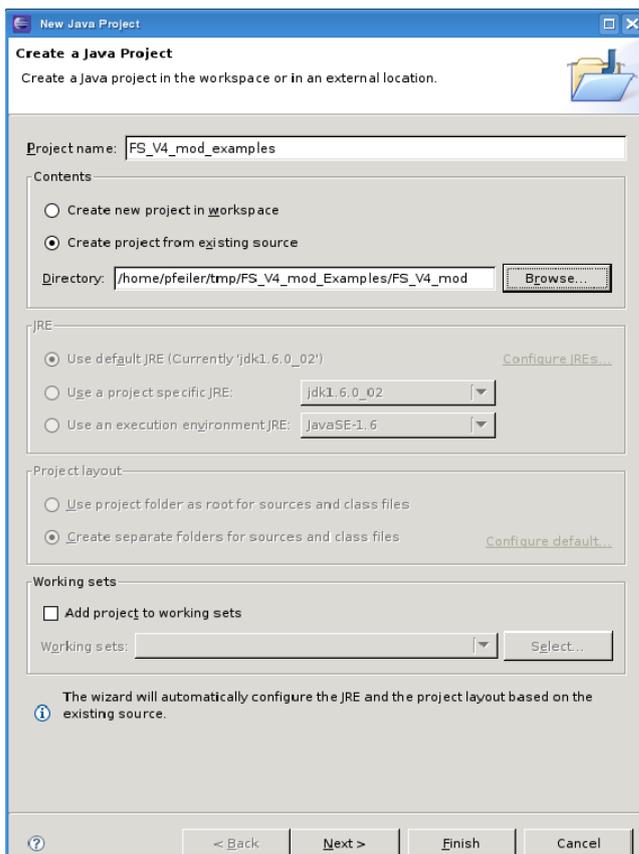
3.10.2 Einrichten der IDE

Die FirstSpirit Beispiel-Modul-Suite² in die Eclipse IDE importieren.

1. Neues Java-Projekt erstellen



2. Projekt aus den Beispiel-Quellen erstellen



² [10] FirstSpirit Modul-Beispiele auf Basis von Eclipse .classpath, Copyright e-Spirit AG



Es wird ein komplett compile-fähiges Projekt erstellt mit allen nötigen Archiven im Classpath. Um ein FirstSpirit-Modul-Archiv (*.fsm) zu erstellen, muss vorher `. setenv.sh` unter Unix oder `setenv.bat` unter Windows ausgeführt werden, um verschiedene Umgebungsvariablen zu setzen. In dem Script müssen folgende Umgebungsparameter auf das lokale Environment angepasst werden.

```
JAVA_HOME=/opt/java/jdk1.6.0
CMS_SERVER_HOME=/opt/firstspirit4
PROJECT_HOME=/home/Me/workspace/FS_V4_mod
```

Listing 15: Anpassen der Umgebungsvariablen für das Eclipse Modul Project Beispiel

Jedes Beispiel-Modul hält seine eigene `build.xml`. Über das Ant-Target `ant assemble-fsm` wird das jeweilige FSM gebaut. Weitere Information zur Integration eines Editors in den FirstSpirit JavaClient finden sich in der README-Datei, welche jedes Modul-Verzeichnis beinhaltet. Bereits erstellte FSM-Archive sind in den Verzeichnissen `target/fsm` eines Moduls zu finden.

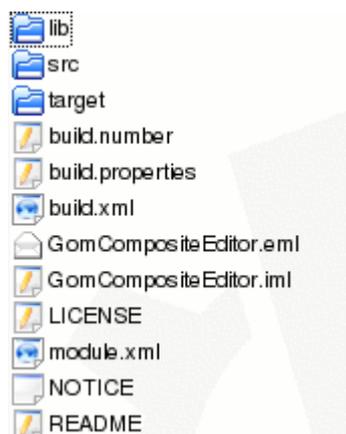


Abbildung 3-1: Modul-Beispiel Build-Verzeichnisstruktur

Die IDE Netbeans kann Eclipse-Projekte importieren, hierzu ist das Plugin Eclipse-Project-Importer erforderlich, diese kann über die Netbeans-Plugin-Verwaltung installiert werden. Um die Beispiel in IntelliJ zu nutzen, können die Quellen nach dem entpacken des Archivs einfach an eine beliebige Stelle ins Dateisystem kopiert werden und anschließend aus der DIE geladen werden über die IntelliJ (*.iml/*ipr) Projekt-Datei.



3.11 GOM – FirstSpirit GUI Object Model

Das FirstSpirit GUI Object Model definiert die grundlegende Architektur und Funktionalität für die Implementierung neuer sowie die Nutzung bestehender (FirstSpirit-eigener) Eingabekomponenten (SwingGadgets, Kapitel 2.9.1.2 Seite 19). SwingGadgets werden in Java implementiert und verfügen über grafische Oberflächenelemente (Swing), wie z.B. Buttons oder Eingabefelder, und funktionale Aspekte (siehe Kapitel 3.12.2 Seite 69), beispielsweise das Bearbeiten und Speichern von Werten. In der Regel werden FirstSpirit-Eingabekomponenten entwickelt, um dem Redakteur neue Funktionalitäten zur Verfügung zu stellen, projektspezifische Aufgaben zu vereinfachen oder auf neue Anforderungen im Betrieb zu reagieren.

SwingGadgets nutzen für die Integration der Eingabekomponente in den FirstSpirit-JavaClient das FirstSpirit GUI Object Model, kurz GOM³. Hierbei kann über die Definition eines XML-Identifiers, z.B. `CUSTOM_TEXTAREA` (Beispiel siehe Kapitel 3.14.2 Seite 127) in der GOM-Form (`de.espirit.firstspirit.access.store.templatestore.gom.GomFormElement`), die Komponente im Vorlagenbereich des FirstSpirit-JavaClients eingebunden werden (im Registerbereich „Formular“) (siehe auch Kapitel 3.27 Seite 212). Ein Anwendungsbeispiel für die GOM-Form einer einfachen SwingGadget-Komponente ist im Kapitel 3.14.2 ab Seite 127 zu finden.



Bei der Vergabe des Identifiers sollte darauf geachtet werden keine Notation mit „FS_“ zu verwenden, da dies zu Konflikten mit FirstSpirit-Eingabekomponenten führen kann.

Prinzipiell erfolgt eine Unterscheidung zwischen:

- GOM-primitiven Elementen und
- Listen GOM-primitiver Elemente.

Zusätzlich gibt es eine:

- Spezialisierung von Listen, die zur Verarbeitung von Elementgruppen dient. Eine Elementgruppe definiert sich dadurch, dass sie GOM-Elemente (`GomLabel`, `GomRadiobutton`) beinhaltet.

³ GOM – FirstSpirit GUI Object Model



Da es in diesem Dokument um die Entwicklung eigener, spezialisierter Komponenten geht, wird auf die Vielzahl der in FirstSpirit bereits vorhandenen Eingabekomponenten nicht näher eingegangen. Dieses Kapitel beschränkt sich auf die grundlegende Verwendung von abstrakten GOM-Elementen für die Implementierung eigener Eingabekomponenten und deren GOM. Weiterführende Informationen zum GUI Object Model und den Standard-Eingabekomponenten in FirstSpirit finden sich in der „Online Dokumentation für FirstSpirit – ODFS“⁴ im Kapitel *Vorlagenentwicklung / Formulare*.

Ein GOM-Element hat oft ein Swing-Pendant, auf das es durch die SwingGadget-Klasse abgebildet wird.

```
1.     ...
2.     for (final GomSearchField searchField : searchFields) {
3.         final int row = tableLayout.getNumRow();
4.         tableLayout.insertRow(row, TableLayoutConstants.PREFERRED);
5.
6.         final JLabel label = new JLabel(searchField.getName());
7.         topPanel.add(label, "0," + row);
8.
9.         final JTextField textField = new JTextField();
10.        _searchFields.put(searchField.getId(), textField);
11.        topPanel.add(textField, "1," + row);
12.    }
13.    ...
```

3.11.1 Typisierung und Mapping

Grundsätzlich definieren Elemente die für sie zulässigen Attribute und Tags über entsprechende Methoden. Dabei entspricht der Methodenname dem Attribut- bzw. Tagnamen mit dem Präfix „get“. Einzige Ausnahme sind Elemente, die `GomList` implementieren. Die für eine Liste zulässigen Elemente werden von der Listenimplementierung selbst als Mapping geliefert (Kapitel 3.11.1.2 Seite 60).

```
1.     public class GomSearchField extends AbstractGomElement {
2.
3.         public static final String TAG = "FIELD";
```

⁴ [4] ODFS - Online Dokumentation für FirstSpirit



```
4.
5.     private String _name;
6.     private Integer _id;
7.
8.
9.
10.
11.     public GomSearchField() {
12.
13.     }
14.
15.
16.
17.
18.     public GomSearchField(final String name, final Integer id) {
19.         _name = name;
20.         _id = id;
21.     }
22.
23.
24.
25.
26.     @GomDoc(description = "Search field name", since = "4.0")
27.     public String getName() {
28.         return _name;
29.     }
30.
31.
32.
33.
34.     public void setName(final String name) {
35.         _name = name;
36.     }
37.
38.
39.
40.
41.     @GomDoc(description = "Search field id", since = "4.0")
42.     public Integer getId() {
43.         return _id;
44.     }
45.
46.
```



```
47.  
48.  
49.     public void setId(final Integer id) {  
50.         _id = id;  
51.     }  
52.  
53.  
54.     //--- GomAbstractFormElement ---//  
55. 

---

56.  
57.  
58.     @Override  
59.     protected String getDefaultTag() {  
60.         return TAG;  
61.     }  
62. 

---

63.  
64.  
65.  
66.     public void validate() throws IllegalStateException {  
67.         super.validate();  
68.     }  
69. }
```

Listing 16: GOM-Typisierung und Mapping

Dieses Beispiel definiert eine Klasse `GomSearchField`. Folgender Tag kann auf diese Klasse abgebildet werden:

```
<FIELD name="street" id="0"/>
```



Wichtig ist hierbei, dass für jedes Attribut eine korrespondierende Methode gleichen Namens mit dem Präfix „get“ und großgeschriebenem ersten Buchstaben existieren muss. Die Methoden müssen immer parameterlos sein. Der Rückgabebetyp entspricht dem erwarteten Attributwert (hier `String` und `Integer` nicht `int`). Jede dieser Methoden muss mit der FirstSpirit-eigenen Gom-Annotation markiert werden. Beispiel: `@GomDoc(description = "Search field name", since = "4.0")`.



3.11.1.1 Direkte Verwendung

Bei direkter Verwendung von Kindelementen muss die beinhaltende Klasse eine entsprechende Methode implementieren, deren Name in Groß-/Kleinschreibung (CamelCase) dem Tagnamen entsprechen muss. Der (konkrete, instanziierbare) Rückgabebetyp bestimmt die validen Tag-Inhalte.

```
1. private static final String TAG = "ADDRESS";
2. private static final String ENTRY_TAG = GomSearchField.TAG;
3.
4. @GomDoc( description="Simple form element for address input data
   (street)." since="4.0" )
5. public class AddressInputData implements AbstractGomElement {
6.
7.     @GomDoc( description="The street input field" since="4.0" )
8.     public GomSearchField getField () {
9.         return ENTRY_TAG;
10.    }
11. }
```

Listing 17: GOM – direkte Verwendung von Kindelementen

XML-Mapping:

```
<ADDRESS>
  <FIELD name="street" id="0"/>
</ADDRESS>
```

Dabei wird der Tagname `FIELD` auf die Methode `getField()` abgebildet und die Attribute des Tags wie oben auf die Methoden von `GomSearchField`.



Der Name des umschließenden Tags `ADDRESS` ist für die implementierende Klasse (wie auch zuvor schon) ohne Bedeutung. Er wird in dem das Tag umschließenden Element für die Abbildung des Tags auf die Implementierungsklasse benötigt.

3.11.1.2 Mengenwertige Verwendung (Elementlisten)

Bei mengenwertiger Verwendung muss die beinhaltende `GUIList`-Implementierung ein entsprechendes Mapping bereitstellen.

Beispiel (schematisch): Realisierung einer einfachen Liste von sprachabhängigen



Informationsobjekten. Die Implementierung definiert lediglich ein Tag als zulässiges Element. Für eine Erweiterung bzgl. der zu speichernden Informationsobjekte muss eine abgeleitete Klasse lediglich die Methode `getGuiElementMappings()` überschreiben.

Die Methode `get()` wird nicht annotiert. Die Annotation für die Liste erfolgt bei der `get`-Methode der Listenklasse und die Annotation der Listenelemente erfolgt als Annotation der jeweilig zugewiesenen Klasse auf Typebene.

```
1.
2.     public class GomSearchFields extends AbstractGomList<GomSearchField> {
3.
4.         private static final String TAG = "SEARCH_FIELDS";
5.         private static final String ENTRY_TAG = GomSearchField.TAG;
6.
7.
8.
9.
10.        public GomSearchFields() {
11.
12.        }
13.
14.
15.
16.        public GomSearchFields(final List<Pair<String, Integer>> fields) {
17.            for (final Pair<String, Integer> field : fields) {
18.                add(new GomSearchField(field.getKey(), field.getValue()));
19.            }
20.        }
21.
22.
23.
24.
25.        @Override
26.        protected String getDefaultTag() {
27.            return TAG;
28.        }
29.
30.
31.
32.        @Override
33.        public Map<String, Class<? extends GomElement>>
getGomElementMappings() {
```



```
34.         final Map<String, Class<? extends GomElement>> mappings =
super.getGomElementMappings();
35.         mappings.put(ENTRY_TAG, GomSearchField.class);
36.         return mappings;
37.     }
38.
39.
40.
41.
42.     public Map<String, Integer> getValues() {
43.         final Map<String, Integer> result = new HashMap<String,
Integer>();
44.         for (final GomSearchField param : this) {
45.             result.put(param.getName(), param.getId());
46.         }
47.         return result;
48.     }
49. }
```

Listing 18: GOM Mengenwertige Verwendung (Elementlisten)

3.11.2 Annotationen

Zur automatischen Verarbeitung der GuiXML (Menge alle Gom-Elemente z.B. eines Absatz-Typs), zur Interpretation als auch zur Dokumentation werden verschiedene Annotationen verwendet.

GomDoc: Die GomDoc-Annotation dient zur Beschreibung anwendungsrelevanter Typen und Methode für die automatische Dokumentation. Sie ermöglicht die Angabe eines Kommentars und einer Bereitstellungs-Version.

```
@GomDoc(description="The name of the street text field",
since="4.0")
```

Mandatory: Indikator, um bestimmte Methoden als zwingend erforderlich zu markieren.

```
@Mandatory
```

InheritAnnotations: Annotation zum Markieren von Stellen, an denen eine Vererbung der Annotation gewünscht ist. Die Vererbung der Annotation wird vom FirstSpirit Annotations-Prozessor geleistet.

```
@InheritAnnotations
```



3.11.3 Abstrakte GOM-Klassen

Die JavaDoc zu den nachfolgenden abstrakten GOM-Klassen befindet sich in der FirstSpirit Access-API⁵-Dokumentation im Paket

```
de.espirit.firstspirit.templatestore.gom.
```

Alle anwendungsrelevanten Methoden müssen mit der FirstSpirit GomDoc-Notation versehen sein. Diese Objekte bzw. deren XML-Repräsentation wird dann vom GOM-Parser bean-ähnlich über Reflection initialisiert. Für spezialisierte GOM-Elemente wird in den meisten Fällen wohl die Klasse **AbstractGomElement** implementiert, welche grundlegende Funktionalitäten für die meisten GOM-Elemente zur Verfügung stellt.

3.11.3.1 AbstractGomComboBox

Die `AbstractGomComboBox`-Klasse stellt die meisten Funktionalitäten zur Verfügung, die eine Combobox nutzt. Ein Anwendungsfall für die Erweiterung dieser abstrakten Klasse ist beispielsweise eine automatische Vervollständigung (Auto-Completion). Dieses lässt sich ohne die Anpassung des GOMs realisieren, d.h. hier ist reine Swing-Funktionalität erforderlich. Sollen jedoch weitere Attribute oder Elemente für die GOM XML-Definition einer Eingabekomponente zu Verfügung gestellt werden, um diese dann ggf. in der Implementierung auszuwerten, müssen diese in der GOM-Definitions-Klasse der Komponente erstellt werden. Ein Anwendungsfall für das konkrete Beispiel der Auto-Completion-Combobox wäre hier, dem Redakteur die Möglichkeit zu geben, über die Gom-Form-Definition zu entscheiden, ob eine Combobox mit der Completion-Funktionalität ausgestattet sein soll oder nicht.

```
1. package de.espirit.firstspirit.opt.examples.gom.combobox;
2.
3. import
  de.espirit.firstspirit.access.store.templatestore.gom.AbstractGomCombobox;
4. import
  de.espirit.firstspirit.access.store.templatestore.gom.TextGomFormElement;
5. import de.espirit.firstspirit.access.store.templatestore.gom.YesNo;
6. import de.espirit.firstspirit.access.editor.ComboboxEditorValue;
7. import de.espirit.firstspirit.common.text.Designator;
8. import de.espirit.common.GomDoc;
9. import de.espirit.common.Default;
```

⁵ [2] Access-API, <http://www.FirstSpirit.de/>, Copyright e-Spirit AG



```
10.
11.
12.  /**
13.   * $Date$
14.   *
15.   * @version $Revision$
16.   */
17.  @GomDoc(description = "GomCombobox form element.", since = "4.1")
18.  public class GomCombobox extends AbstractGomCombobox implements
    TextGomFormElement {
19.
20.     // same TAG defined in the module descript. @see module.xml
    <name>CMS_EXAMPLE_GOMCOMBOBOX</name>
21.     public static final String TAG = "CMS_EXAMPLE_GOMCOMBOBOX";
22.
23.     private String _name;
24.     private Integer _id;
25.     private YesNo _autocompletion;
26.     private YesNo _editable;
27.     private YesNo _extAutoCompletion;
28.
29.
30.
31.
32.     public GomCombobox() {
33.         super();
34.     }
35.
36.
37.
38.
39.     public GomCombobox(final String name, final Integer id) {
40.         _name = name;
41.         _id = id;
42.     }
43.
44.
45.
46.
47.     public void setName(final String name) {
48.         _name = name;
49.     }
50.
```



```
51.
52.
53.
54.     @GomDoc(description = "The Combobox id", since = "4.1")
55.     public Integer getId() {
56.         return _id;
57.     }
58.
59.
60.
61.
62.     public void setId(final Integer id) {
63.         _id = id;
64.     }
65.
66.
67.
68.
69.     @GomDoc(description="Editable indicator.", since="4.1")
70.     @Default("NO")
71.     public YesNo getEditable() {
72.         return _editable;
73.     }
74.
75.
76.
77.
78.     public void setEditable(final YesNo editable) {
79.         _editable = editable;
80.     }
81.
82.
83.
84.
85.     @GomDoc(description = "Enable the auto completion feature of the
86.     combobox.", since = "4.1")
87.     @Default("NO")
88.     public YesNo getAutoCompletion() {
89.         return _autocompletion;
90.     }
91.
92.
```



```
93.
94.     public void setAutoCompletion(final YesNo enable) {
95.         _autoCompletion = enable;
96.     }
97.
98.
99.
100.
101.     @GomDoc(description = "Enable the extended combobox auto
102. completion mechanism.", since = "4.1")
103.     @Default("NO")
104.     public YesNo getExtAutoCompletion() {
105.         return _extAutoCompletion;
106.     }
107.
108.
109.
110.     public void setExtAutoCompletion(final YesNo enable) {
111.         _extAutoCompletion = enable;
112.     }
113.
114.
115.
116.     /**
117.      * Return the default tag for a gom element.
118.      *
119.      * @return The elements default tag.
120.      */
121.     @Override
122.     protected String getDefaultTag() {
123.         return TAG;
124.     }
125.
126.
127.
128.     @Override
129.     public void validate() throws IllegalStateException {
130.         super.validate();
131.     }
132. }
```

Listing 19: GOM Beispiel Combobox



```
1. <CMS_EXAMPLE_GOMCOMBOBOX
2.     name="gomcombobox2"
3.     autoComplete="no"
4.     editable="yes"
5.     extAutoCompletion="yes"
6.     length="10"
7.     useLanguages="yes">
8.     <LANGINFOS>
9.         <LANGINFO lang="*" label="GOM ExtAuto-Completion-Combobox
Example"/>
10.        <LANGINFO lang="DE" label="GOM ExtAuto-Completion-Combobox
Beispiel"/>
11.        <LANGINFO lang="EN" label="GOM ExtAuto-Completion-Combobox
Example"/>
12.     </LANGINFOS>
13. </CMS_EXAMPLE_GOMCOMBOBOX>
```

Listing 20: GOM-Form Representation



Ein vollständiges Beispiel der Combobox-Komponente ist unter Eclipse Modul Suite ⁶ zu finden.

3.11.3.2 AbstractGomElement

Die abstrakte Klasse `AbstractGomElement` implementiert gemeinsame Funktionalitäten für das GuiXML-Handling.

3.11.3.3 AbstractGomFormElement

Diese Klasse implementiert alle grundlegenden Funktionalitäten für das FirstSpirit GuiXML-Handling und kann von fast jeder zu entwickelnden Editoren-Komponente genutzt werden; um eigene, spezialisierte Editoren zu implementieren (Beispiel siehe Kapitel 3.14.2 Seite 127).

⁶ [10] FirstSpirit Modul-Beispiele auf Basis von Eclipse .classpath, Copyright e-Spirit AG





Ein vollständiges Beispiel der Combobox-Komponente ist unter [MODDEV40_Sources\EclipseModulSuite](#) zu finden.

3.11.3.4 AbstractGomList<T extends GomElement>

Abstrakte Implementierung einer typisierten Liste, die andere GOM-Elemente aufnehmen kann (Kapitel 3.11.1.2 Seite 60).

Die Implementierung in diesem Bereich ist noch nicht vollständig abgeschlossen. Die Dokumentation erfolgt sobald wie möglich.

3.11.3.5 AbstractGomSelect

Abstrakte Radiobutton-Implementierung.

Die Implementierung in diesem Bereich ist noch nicht vollständig abgeschlossen. Die Dokumentation erfolgt sobald wie möglich.



3.12 Von Gadgets, Aspects, Brokern und Agents

Dieses Kapitel enthält einen Überblick über die wichtigsten Implementierungen und Schnittstellen des Komponenten-Modells in FirstSpirit 5 und erläutert einige grundlegende Begriffe.

3.12.1 Gadgets

Ein **SwingGadget** ist die grafische Repräsentation einer Eingabekomponente in der FirstSpirit-Redaktionsumgebung (JavaClient) – das Pendant zu den GuiEditoren der Version 4. SwingGadgets werden in Java implementiert und verfügen über grafische Oberflächenelemente (Swing), wie z.B. Buttons oder Eingabefelder, und funktionale Aspekte (siehe Kapitel 3.12.2 Seite 69), wie z.B. das Speichern von Werten. Zudem werten SwingGadgets Benutzeraktionen und Änderungen aus und leiten diese an die visuelle Darstellung der Komponente weiter.

Alle SwingGadget-Implementierungen müssen das Interface SwingGadget (Basistyp) implementieren.

Ein Beispiel für die SwingGadget-Implementierung einer einfachen Eingabekomponente befindet sich in Kapitel 3.14.4, Seite 130.

3.12.2 Aspekte (SwingGadget)

Jedes SwingGadget kann eine Vielzahl funktionaler Aspekte („Aspects“) implementieren. Die Funktionen „editierbar“ und „vergrößerbar“ einer Eingabekomponente stellen beispielsweise funktionale Aspekte dar. Diese Kernfunktionalitäten müssen im neuen FirstSpirit-Komponentenmodell nicht von der SwingGadget-Implementierung selber umgesetzt werden, sondern stehen als Interface (Aspekt-Typen) zur Verfügung.

Dabei unterscheidet man:

- **Standardaspekte**, die der neuen Komponente nicht explizit hinzugefügt werden müssen, sondern automatisch durch die Erweiterung der SwingGadget-Klasse mit der abstrakten Basis-Implementierung `AbstractValueHoldingSwingGadget<T, F extends GomFormElement>` verfügbar sind und
- **Optionale Aspekte**, die der neuen Komponente explizit über den Aufruf `addAspect(@NotNull AspectType<A> type, @NotNull A aspect)` im Konstruktor der SwingGadget-Klasse hinzugefügt werden können.



Über Aspekte kann ein Komponentenentwickler komplexe Funktionalität, beispielsweise Drag&Drop, sehr einfach bei der Entwicklung neuer Eingabekomponenten verwenden. Er muss dazu lediglich das gewünschte Interface (bzw. die abstrakte Basis-Implementierung) einbinden und die dort gestellten Anforderungen an die Implementierung erfüllen. Das umliegende FirstSpirit-Gadget-Framework behandelt dann automatisch alle weiteren Funktionen, beispielsweise das Drop-Handling für eine Eingabekomponente.

STANDARD-ASPEKTE

Sofern eine Eingabekomponente das **Bearbeiten und Speichern von Werten** ermöglichen soll, werden die Aspekte `Editable` und `ValueHolder` benötigt:

- `ValueHolder`: Speichern von Werten (siehe Kapitel 3.12.2.3 Seite 73).
- `Editable`: Bearbeiten von Werten (siehe Kapitel 3.12.2.4 Seite 74).

Ein Beispiel zur Verwendung der Aspekte `Editable` und `ValueHolder` in eine `SwingGadget`-Implementierung ist in Kapitel 3.14.4 (ab Seite 130) beschrieben.

Ist das **Einblenden einer Beschriftung (Label)** gewünscht, muss die Eingabekomponente den Aspekt `Labelable` implementieren:

- `Labelable`: Beschriftung von Eingabekomponenten (siehe Kapitel 3.12.2.5 Seite 75).

OPTIONALE ASPEKTE

Abhängig von der Eingabekomponente kann das Hinzufügen weiterer Aspekte zur Anpassung des Layouts sinnvoll sein, beispielsweise für eine **einzeilige Darstellung**, die **Einstellung einer festen Breite**, eine **Größenänderung** durch den Redakteur oder das Bearbeiten in einem **separaten Bearbeitungsfenster**:

- `SingleLineable`: Einzeilige, visuelle Darstellung von Eingabekomponenten (beispielsweise für die Auswahl zwischen zwei vorgegebenen Werten über einen Radiobutton (vgl. `CMS_INPUT_TOGGLE`) (siehe Kapitel 3.12.2.8 Seite 78).
- `Resizable`: Größenänderung einer Eingabekomponente durch den Redakteur über Buttons (+/-) (siehe Kapitel 3.12.2.9 Seite 79).
- `SeparateEditable`: Öffnen der Eingabekomponente in einem separaten, großen Bearbeitungsfenster über einen Button (vgl. `CMS_INPUT_DOM`) (siehe Kapitel 3.12.2.10 Seite 80).
- `WidthFixable`: Einstellung einer festen Breite für die Eingabekomponente (siehe Kapitel 3.12.2.11 Seite 81).



In einzeiligen Eingabekomponenten kann es erwünscht sein, die **Beschriftung einer Eingabekomponente ein- oder auszublenden**:

- `LabelHideable`: Ausblenden der Beschriftung einer Eingabekomponente (siehe Kapitel 3.12.2.6 Seite 76).

Eine Eingabekomponente kann **Inhalte hervorheben und markieren**. Über Aspekte kann auf die unterschiedlichen Markierungs-Ereignisse individuell reagiert werden.

- `SwingFocusable`: Markierungsart FOCUS für die Umrahmung innerhalb des Content-Hightlightings (siehe Kapitel 3.12.2.12 Seite 82).
- `Highlightable`: Markierungsart MATCH für die Treffermarkierung bei der Suche (siehe Kapitel 3.12.2.13 Seite 83).
- `DifferenceVisualizing`: Markierungsarten INSERTED, DELETED und CHANGED für die Markierung innerhalb der Differenz-Visualisierung (siehe Kapitel 3.12.2.14 Seite 84)

Um das **Speichern von Änderungen** möglichst effizient zu gestalten, führt das FirstSpirit-Gadget-Framework bei jeder Änderung innerhalb einer Eingabekomponente einen Vergleich des zuletzt gespeicherten Wertes mit dem aktuell im SwingGadget enthaltenen Wert durch. Abhängig von der Art der Eingabekomponente und des zugehörigen Daten-Container-Typs kann dieser Vergleich unvorteilhaft sein. Eigene Methoden zur **Änderungserkennung** können über die funktionalen Aspekte `ChangeManaging` und `ValueLikening` implementiert werden.

- `ChangeManaging`: Erkennen inhaltlicher Änderungen innerhalb einer Eingabekomponente. Dieser Aspekt sollte verwendet werden, wenn eine Transformation des inneren Modells der Eingabekomponente in den Persistenztyp *teuer ist* (siehe Kapitel 3.12.2.15 Seite 85).
- `ValueLikening`: Erkennen inhaltlicher Änderungen innerhalb einer Eingabekomponente. Dieser Aspekt sollte verwendet werden, wenn eine Transformation des inneren Modells in den Persistenztyp *nicht möglich ist* (siehe Kapitel 3.12.2.16 Seite 87).

Ein Beispiel zur Verwendung der Aspekte `ChangeManaging` und `ValueLikening` in einer `SwingGadget`-Implementierung ist in Kapitel 3.14.5 (ab Seite 133) beschrieben.

Die Aspekte `TransferHandling` und `TransferSupplying` ermöglichen das **Übertragen typisierter Objekte per Drag & Drop**. Die `SwingGadget`-Komponente muss dazu lediglich als Anbieter oder Empfänger für den Drag&Drop-Transfer registriert werden. Das eigentliche Drag&Drop-Handling wird dann versteckt vom FirstSpirit-Framework ausgeführt.



- `TransferSupplying`: Liefern eines Objekts per Drag & Drop (siehe Kapitel 3.12.2.18 Seite 90)
- `TransferHandling`: Empfangen eines Objekts per Drag & Drop (siehe Kapitel 3.12.2.19 Seite 92)

Weitere Aspekte:

- `DisplaySettingsAware`: Umschalten der Anzeigesprache (siehe Kapitel 3.12.2.7 Seite 77).
- `IntegrityValidating`: Validiert die Datenintegrität des Persistenztyps (siehe Kapitel 3.12.2.17 Seite 88)
- `ShowUsagesOperation`: kein Aspekt – zuständig für Operations (n.dok.)

Die unterschiedlichen Aspekt-Typen stellen spezielle Anforderungen an die Implementierung der SwingGadget-Komponente, z.B. in Form von Rückgabewerten/-Typen dar, die nachfolgend erläutert werden.

3.12.2.1 Das Interface Aspectable

Die Implementierung in diesem Bereich ist noch nicht vollständig abgeschlossen. Die Dokumentation erfolgt sobald wie möglich.

3.12.2.2 Die abstrakte Klasse AspectType<T> extends TokenType<T>

Die Implementierung in diesem Bereich ist noch nicht vollständig abgeschlossen. Die Dokumentation erfolgt sobald wie möglich.



3.12.2.3 Aspekt: ValueHolder<T>

Aspect: ValueHolder<T>**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Abhängig von der SwingGadget-Implementierung kann eine Eingabekomponente die Fähigkeit haben, Werte zu speichern. Nach der Umstellung der Komponentenentwicklung auf SwingGadgets werden die EditorValues (siehe Kapitel 3.12.6.1 Seite 97) nicht mehr direkt an die Implementierung der Eingabekomponente übergeben (wie in FirstSpirit 4), sondern intern nur noch über den typisierten Aspekt `ValueHolder<T>` behandelt, wobei der Parameter Typ `<T>` immer der Daten-Container-Typ des `ValueEngineer<T>` ist (siehe Kapitel 3.12.6.3 Seite 98) und zum Wertetyp der zugehörigen SwingGadget-Implementierung passen muss. Das umliegende FirstSpirit-Gadget-Framework behandelt dann alle weiteren Funktionen, beispielsweise das persistente Speichern des Wertes.

Dieser Aspekt kann der SwingGadget-Implementierung entweder einzeln (über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements ValueHolder<T> {
2.
3. ...
4.
5.     public mySwingGadget(final
6.         SwingGadgetContext<myGomForm> context) {
7.         super(context);
8.         // type-safe value holder registration
9.         addAspect(ValueHolder.TYPE, this);
10.    }
11.    ...
12.
13. }
```

oder wird bei Erweiterung der abstrakte Basisimplementierung

`AbstractValueHoldingSwingGadget<T, F extends GomFormElement>`
geerbt:

```
1. public class mySwingGadget extends
2.     AbstractValueHoldingSwingGadget<myValueType, myGomForm>{
3. ...
4. }
```

Beispiel zur Verwendung des Aspekts siehe Kapitel 3.14.4 (Seite 130).



3.12.2.4 Aspekt: Editable

Aspect: Editable
Package: de.espirit.firstspirit.ui.gadgets.aspects

Abhängig von der SwingGadget-Implementierung kann eine Eingabekomponente die Fähigkeit haben, Werte zu bearbeiten und persistent zu speichern. Der Aspekt Editable markiert eine Eingabekomponente als editierbar und ermöglicht so, das Sperren und Entsperrern der Eingabekomponenten zum Bearbeiten der darin enthaltenen Werte. Zuständig für das Setzen und Auslesen der Werte ist der typisierte Aspekt ValueHolder<T> (siehe Kapitel 3.12.2.3 Seite 73). Der Wert vom Typ T des ValueHolder<T> kann nach dem Entsperrern der Eingabekomponente angepasst (neu gesetzt oder gelöscht) werden.

Dieser Aspekt kann der SwingGadget-Implementierung entweder einzeln (über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements ValueHolder<T>,
   Editable {
2.
3. ...
4.
5.     public mySwingGadget(final
   SwingGadgetContext<myGomForm> context) {
6.         super(context);
7.         // type-safe value holder registration for
   write-enabled access
8.         addAspect(ValueHolder.TYPE, this);
9.         addAspect(Editable.TYPE, this);
10.    }
11.
12.    ...
13.
14.    }
```

oder wird bei Erweiterung der abstrakte Basisimplementierung

`AbstractValueHoldingSwingGadget<T, F extends GomFormElement>`
geerbt:

```
1. public class mySwingGadget extends
   AbstractValueHoldingSwingGadget<myValueType, myGomForm>{
2. ...
3. }
```

Beispiel zur Verwendung des Aspekts siehe Kapitel 3.14.4 (Seite 130).



3.12.2.5 Aspekt: Labelable

Aspect: Labelable**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Eine Eingabekomponente kann eine Beschriftung (Label) besitzen, die redaktionelle Hinweise für die Verwendung der Eingabekomponente enthalten kann (beispielsweise die Beschriftung „Überschrift“ für eine Text-Eingabekomponente). Zuständig für das Anzeigen der Beschriftung ist der Aspekt Labelable.



Der Aspekt Editable erbt von Labelable (`public interface Editable extends Labelable {...}`). Sollen beide Aspekte verwendet werden, muss Labelable aber dennoch gesondert als Aspekt registriert werden.

Soll eine Eingabekomponente eine Beschriftung erhalten, muss sie den Aspekt Labelable implementieren.

Dieser Aspekt kann der SwingGadget-Implementierung entweder einzeln (über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
4. public class mySwingGadget implements Labelable {
5.
6. ...
7.
8.     public mySwingGadget(final
           SwingGadgetContext<myGomForm> context) {
9.         super(context);
10.            addAspect(Labelable.TYPE, this);
11.        }
12.
13.     ...
14.
15.    }
```

oder wird bei Erweiterung der abstrakte Basisimplementierung

`AbstractValueHoldingSwingGadget<T, F extends GomFormElement>`
geerbt:

```
1. public class mySwingGadget extends
   AbstractValueHoldingSwingGadget<myValueType, myGomForm>{
2. ...
3. }
```



3.12.2.6 Aspekt: LabelHidable

Aspect: LabelHidable**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Eine Eingabekomponente kann eine Beschriftung besitzen (vgl. Kapitel 3.12.2.6 Seite 76). In einzeiligen Eingabekomponenten kann diese Beschriftung mithilfe des Aspekts LabelHidable über einen Parameter ein- oder ausgeblendet werden.

Das parametrisierte Ein- bzw. Ausblenden einer Beschriftung ist nur für SwingGadgets möglich, die ein Label verwenden (also den Aspekt Labelable implementieren) (siehe Kapitel 3.12.2.6) und eine einzeilige Darstellung über den Aspekt SingleLineable (siehe Kapitel 3.12.2.8) unterstützen.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements Labelable, LabelHidable,
   SingleLineable {
2.
3. ...
4.
5.     public mySwingGadget(final
       SwingGadgetContext<myGomForm> context) {
6.         super(context);
7.         addAspect(Labelable.TYPE, this);
8.         addAspect(LabelHidable.TYPE, this);
9.         addAspect(SingleLineable.TYPE, this);
10.    }
11.
12.    ...
13.
14.    }
```

Des Weiteren muss die Methode `boolean isLabelHiding()` implementiert werden, die definiert, ob eine Beschriftung angezeigt wird oder nicht. Das entsprechende Flag kann abhängig davon gesetzt werden, ob innerhalb der Eingabekomponente ein Parameter für die Anzeige der Beschriftung konfiguriert wurde. Ein entsprechender Parameter kann z.B. in der GomForm-Implementierung des SwingGadgets zugefügt werden (Beispiel für das Hinzufügen von Parametern siehe Kapitel 3.14.2 Seite 127).



3.12.2.7 Aspekt: DisplaySettingsAware

Aspect: DisplaySettingsAware
Package: de.espirit.firstspirit.ui.gadgets.aspects

Abhängig von der SwingGadget-Implementierung kann eine Eingabekomponente die Fähigkeit haben sprachabhängige Beschriftungen anzuzeigen. Diese sprachabhängigen Beschriftungen werden in der XML-Definition einer Eingabekomponente im Vorlagenbereich innerhalb der Tags <LANGINFOS></LANGINFOS> definiert. Abhängig von der gewählten Anzeigesprache werden die entsprechenden sprachabhängigen Beschriftungen der Komponente (Eingabefelder, Tooltips, Elemente einer Combobox, usw.) eingeblendet. Zuständig für das Einblenden einer sprachabhängigen Beschriftung ist der Aspekt DisplaySettingsAware.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements DisplaySettingsAware {
2.
3. ...
4.
5.     public mySwingGadget(final
6.         SwingGadgetContext<myGomForm> context) {
7.         super(context);
8.         addAspect(DisplaySettingsAware.TYPE, this);
9.     }
10.
11.     ...
12. }
```

Des Weiteren muss die Methode `void setDisplayLanguage(final Language language)` implementiert werden, über die eine Anzeigesprache für die Anzeige der sprachabhängigen Beschriftungen festgelegt wird.



3.12.2.8 Aspekt: SingleLineable

Aspect: SingleLineable**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Abhängig von der SwingGadget-Implementierung kann es sinnvoll sein, die visuelle Darstellung einer Komponente einzeilig zu gestalten. Mithilfe des Aspekts SingleLineable kann festgelegt werden, ob eine Eingabekomponente in einer Zeile (ohne Rahmen) dargestellt werden soll oder nicht.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements SingleLineable {
2.
3. ...
4.
5.     public mySwingGadget(final
        SwingGadgetContext<myGomForm> context) {
6.         super(context);
7.         addAspect(SingleLineable.TYPE, this);
8.     }
9.
10.     ...
11.
12. }
```

Des Weiteren muss die Methode `boolean isSingleLine()` implementiert werden, die definiert, ob eine Beschriftung angezeigt wird oder nicht. Das entsprechende Flag kann abhängig davon gesetzt werden, ob innerhalb der Eingabekomponente ein Parameter (z.B. `singleLine="yes"`) für die einzeilige Darstellung konfiguriert wurde. Ein entsprechender Parameter kann z.B. in der GomForm-Implementierung des SwingGadgets zugefügt werden (Beispiel für das Hinzufügen von Parametern siehe Kapitel 3.14.2 Seite 127).



3.12.2.9 Aspekt: Resizable

Aspect: Resizable**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Abhängig von der SwingGadget-Implementierung kann es sinnvoll sein, dem Redakteur eine vertikale Größenänderung der Eingabekomponente zu ermöglichen. Mithilfe des Aspekts Resizable können zwei Buttons (+/-) innerhalb der Eingabekomponente eingeblendet werden, die eine vertikale Vergrößerung (+) oder Verkleinerung (-) der Komponente realisieren. (Eine horizontale Größenänderung kann über den Aspekt: WidthFixable realisiert werden, siehe Kapitel 3.12.2.11, Seite 81).

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements Resizable
2.
3. ...
4.
5.     public mySwingGadget(final
6.         SwingGadgetContext<myGomForm> context) {
7.         super(context);
8.         addAspect(Resizable.TYPE, this);
9.     }
10.
11.     ...
12. }
```



3.12.2.10 Aspekt: SeparateEditable

Aspect: SeparateEditable**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Abhängig von der SwingGadget-Implementierung kann es sinnvoll sein, das Bearbeiten der Werte innerhalb einer Eingabekomponente in ein separates Fenster auszulagern (vgl. auch Aspekt: Editable, Kapitel 3.12.2.4). Mithilfe des Aspekts SeparateEditable kann ein Button  innerhalb der Eingabekomponente eingeblendet werden, der dem Redakteur gestattet, die Komponente in einem externen Bearbeitungsfenster in voller Bildschirmgröße zu öffnen. Die Inhalte können entweder direkt im externen Fenster gespeichert werden, oder sie werden beim Schließen des Fensters automatisch in die Eingabekomponente übernommen und können dort gespeichert werden.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements SeparateEditable {
2.
3. ...
4.
5.     public mySwingGadget (final
6.         SwingGadgetContext<myGomForm> context) {
7.         super (context);
8.         addAspect (SeparateEditable.TYPE, this);
9.     }
10.
11.     ...
12. }
```



3.12.2.11 Aspekt: WidthFixable

Aspect: WidthFixable**Package:** de.espirit.firstspirit.ui.gadgets.aspects

Die Darstellung einer Eingabekomponente erfolgt zunächst immer mit einer vordefinierten Standardbreite. Über den Parameter `hFill` kann der Vorlagenentwickler ferner definieren, ob eine Eingabekomponente auf der vollen zur Verfügung stehenden Anzeigebreite dargestellt werden soll (`hfill="yes"`) oder nicht. Soll für eine Eingabekomponente eine neue, unveränderliche Breite definiert werden, die nicht der Standardbreite oder der vollen Anzeigebreite entspricht, kann mithilfe des Aspekts `WidthFixable` ein Flag gesetzt werden, das die Eingabekomponente als „fixiert“ definiert. Die gewünschte Breite wird beim Rendern der Komponente von FirstSpirit nun nicht mehr automatisch durch eine andere Breite ersetzt.

Dieser Aspekt kann der `SwingGadget`-Implementierung über die `implements`-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der `SwingGadget`-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements WidthFixable
2.
3. ...
4.
5.     public mySwingGadget (final
6.         SwingGadgetContext<myGomForm> context) {
7.         super(context);
8.         addAspect(WidthFixable.TYPE, this);
9.     }
10.
11.     ...
12. }
```

Des Weiteren muss die Methode `boolean isWidthFixed()` implementiert werden, die definiert, ob eine Eingabekomponente fixiert ist oder nicht. Das entsprechende Flag kann beispielsweise abhängig davon gesetzt werden, ob innerhalb der Eingabekomponente ein Parameter für die Breite konfiguriert wurde. Ein entsprechender Parameter für die Definition der Breite kann z.B. in der `GomForm`-Implementierung des `SwingGadgets` zugefügt werden (Beispiel für das Hinzufügen von Parametern siehe Kapitel 3.14.2 Seite 127).



3.12.2.12 Aspekt: SwingFocusable

Aspect: SwingFocusable**Package:** de.espirit.firstspirit.ui.gadgets.aspects.focus

Eine Eingabekomponente kann die Fähigkeit besitzen, ihre Inhalte zu markieren. Eine bekannte Markierungsart ist die Umrahmung der Komponente bzw. ihres Inhalts über das ContentHighlighting (Markierungsart FOCUS). Mithilfe des Aspekts SwingFocusable kann auf dieses Ereignis (Setzen einer Fokus-Markierung) reagiert werden. Die Eingabekomponente kann die Markierung (Focus) dann beispielsweise an eine geeignete innere Komponente weiterreichen, die darauf entsprechend reagiert. Das gilt auch für den Verlust von Markierungen.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements SwingFocusable {
2.
3. ...
4.
5.     public mySwingGadget(final
        SwingGadgetContext<myGomForm> context) {
6.         super(context);
7.         addAspect(SwingFocusable.TYPE, this);
8.     }
9.
10.     ...
11.
12. }
```

Des Weiteren muss die Methode `void acceptFocus(@NotNull Handler handler)` implementiert werden, die den Fokus auf einer Eingabekomponente setzt. Dazu sollte immer der übergebene Handler (und keine direkten Swing-Mechanismen) verwendet werden:

```
1. public void acceptFocus(@NotNull final Handler
    handler) {
2.     handler.focusOn(myComponent);
3. }
```



3.12.2.13 Aspekt: Highlightable

Aspect: Highlightable**Package:** de.espirit.firstspirit.ui.gadgets.aspects.highlight

Eine Eingabekomponente kann die Fähigkeit besitzen, ihre Inhalte zu markieren. Eine bekannte Markierungsart ist die Treffermarkierung für Suchergebnisse (Markierungsart MATCH). Das Hervorheben der Komponente, die die entsprechenden Suchergebnisse enthält, wird bereits automatisch vom FirstSpirit-Gadget-Framework bereitgestellt. Weitere Funktionen, beispielsweise eine Markierung der Treffer innerhalb der Komponente oder auch das Ändern des Sichtbarkeitsbereichs (Scrolling) zur Anzeige eines Treffers in der Komponente, können vom Komponententwickler über den Aspekt Highlightable hinzugefügt werden. Diese Funktionen können vom FirstSpirit-Gadget-Framework nicht allgemein realisiert werden, da dazu eine genaue Kenntnis des inneren Aufbaus der Komponente erforderlich ist. Mithilfe des Aspekts Highlightable kann die Eingabekomponente auf dieses Ereignis (Setzen einer Match-Markierung) reagieren und die Markierungen (Matches) dann beispielsweise an eine geeignete innere Komponente weiterreichen, die darauf entsprechend reagiert. Das gilt auch, für den Verlust von Markierungen.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements Highlightable {
2. ...
3.     public mySwingGadget(final
           SwingGadgetContext<myGomForm> context) {
4.         super(context);
5.         addAspect(Highlightable.TYPE, this);
6.     }
7. ...
8. }
```

Des Weiteren muss die Methode `public void highlight(List<? extends Match> matches)` implementiert werden, die Infrastruktur zur Verfügung stellt, um den Treffer auch innerhalb der Komponente hervorzuheben und ggf. durch Scrolling sichtbar zu machen.

Beispiel zur Verwendung des Aspekts siehe Kapitel 3.14.9 (Seite 142).



3.12.2.14 Aspekt: DifferenceVisualizing

Aspect: DifferenceVisualizing

Package: de.espirit.firstspirit.ui.gadgets.aspects

Eine Eingabekomponente kann die Fähigkeit besitzen, ihre Inhalte zu markieren. Diese Eigenschaft wird auch für die Markierungen innerhalb der Differenz-Visualisierung (Markierungsarten INSERTED, DELETED und CHANGED) genutzt. Dabei können die Inhalte einer Eingabekomponente (in zwei unterschiedlichen Revisionen) miteinander verglichen werden. Alle Änderungen von einer Revision zu einer anderen Revision werden innerhalb der Eingabekomponente markiert. Mithilfe des Aspekts DifferenceVisualizing kann die Eingabekomponente auf diese Ereignisse (Setzen einer Änderungs-Markierung) reagieren.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements DifferenceVisualizing {
2.
3. ...
4.
5.     public mySwingGadget(final
6.         SwingGadgetContext<myGomForm> context) {
7.         super(context);
8.         addAspect(DifferenceVisualizing.TYPE, this);
9.     }
10.     ...
11.
12. }
```

Die SwingGadget-Eingabekomponente muss anschließend die Methode `visualizeDifferences(final List<Difference> differences)` implementieren. Die erforderliche Liste der Änderungen muss von der Methode `List<Difference> contrastWith(final Language language, final Object other)` innerhalb der EditorValue-Implementierung geliefert werden (da sich die Änderungen auf die in der Eingabekomponente enthaltenen Werte beziehen). Für den Vergleich einfacher Textwerte, kann die Methode der abstrakten Basisimplementierung `AbstractEditorValue<T>` verwendet werden, für komplexere Werte ist es jedoch notwendig, die Methode `contrastWith(...)` in der EditorValue-Implementierung der Eingabekomponente zu überschreiben.



3.12.2.15 Aspekt: ChangeManaging

Aspect: ChangeManaging

Package: de.espirit.firstspirit.ui.gadgets.aspects

Sofern eine Eingabekomponente Werte speichern und bearbeiten kann (vgl. Aspekt: ValueHolder<T> in Kapitel 3.12.2.3 und Aspekt: Editable in Kapitel 3.12.2.4), müssen Änderungen innerhalb der Komponente nach außen propagiert werden. Über den Aufruf von `notifyValueChange(final GadgetIdentifizier identifizier)` teilt das SwingGadget dem äußeren Framework mit, dass sich sein Inhalt geändert hat. Um unnötiges Speichern und die damit verbundene Erzeugung einer neuen Revision eines FirstSpirit-Objekts zu vermeiden, wird bei jeder Änderung im Editor (also nach jedem Aufruf von `notifyValueChange(final GadgetIdentifizier identifizier)`) zunächst überprüft, ob der geänderte Wert dem zuletzt gespeicherten Wert entspricht (Speichern nicht notwendig) oder nicht (Speichern notwendig). Das FirstSpirit-Gadget-Framework führt dazu einen „Equals“-Vergleich des zuletzt gespeicherten Wertes mit dem aktuell im SwingGadget enthaltenen Wert durch, indem der Wert des SwingGadgets über die Methode `getValue()` abgefragt wird.

Der Aspekt ChangeManaging sollte immer dann verwendet werden, wenn ein einfacher „Equals“-Vergleich des zuletzt gespeicherten Wertes mit dem aktuell im SwingGadget enthaltenen Wert mithilfe von `getValue()` zu teuer ist und/oder es eine performantere Implementierung zur Änderungserkennung gibt.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements ChangeManaging {
2. ...
3.     public mySwingGadget (final
4.         SwingGadgetContext<myGomForm> context) {
5.         super (context);
6.         addAspect (ChangeManaging.TYPE, this);
7.     }
8. }
```

Des Weiteren müssen alle Methoden des Interfaces implementiert werden:

- Die Methode `boolean hasChanges()` ermittelt, ob der Inhalt einer Eingabekomponente geändert wurde oder nicht.



- Die Methode `void replaceValue(T value)` ersetzt den aktuellen Inhalt einer Eingabekomponente durch den übergebenen, neuen Wert. Ein nachfolgender Aufruf der Methode `boolean hasChanges()` sollte anschließend `true` zurückliefern.
- Die Methode `void clearValue()` löscht den aktuellen Inhalt einer Eingabekomponente. Ein nachfolgender Aufruf der Methode `boolean hasChanges()` sollte anschließend `true` zurückliefern.

Beispiel zur Verwendung des Aspekts siehe Kapitel 3.14.5.1 (Seite 135).



3.12.2.16 Aspekt: ValueLikening

Aspect: ValueLikening

Package: de.espirit.firstspirit.ui.gadgets.aspects

Sofern eine Eingabekomponente Werte speichern und bearbeiten kann (vgl. Aspekt: ValueHolder<T> in Kapitel 3.12.2.3 und Aspekt: Editable in Kapitel 3.12.2.4), müssen Änderungen innerhalb der Komponente nach außen propagiert werden. Über den Aufruf von `notifyValueChange(final GadgetIdentifizier identifizier)` teilt das `SwingGadget` dem äußeren Framework mit, dass sich sein Inhalt geändert hat. Um unnötiges Speichern und die damit verbundene Erzeugung einer neuen Revision eines FirstSpirit-Objekts zu vermeiden, wird bei jeder Änderung im Editor (also nach jedem Aufruf von `notifyValueChange(final GadgetIdentifizier identifizier)`) zunächst überprüft, ob der geänderte Wert dem zuletzt gespeicherten Wert entspricht (Speichern nicht notwendig) oder nicht (Speichern notwendig). Das FirstSpirit-Gadget-Framework führt dazu einen „Equals“-Vergleich des zuletzt gespeicherten Wertes mit dem aktuell im `SwingGadget` enthaltenen Wert durch, indem der Wert des `SwingGadgets` über die Methode `getValue()` abgefragt wird.

Der Aspekt `ValueLikening` sollte immer dann verwendet werden, wenn eine Transformation in das Persistenzformat (beim Aufruf von `getValue()`) nicht zu jedem Zeitpunkt möglich ist.

Dieser Aspekt kann der `SwingGadget`-Implementierung über die `Implements`-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der `SwingGadget`-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements ValueLikening<T> {
2. ...
3.     public mySwingGadget(final
4.         SwingGadgetContext<myGomForm> context) {
5.         super(context);
6.         addAspect(ValueLikening.TYPE, this);
7.     }
8. }
```

Das Interface `ValueLikening` ist typisiert, d. h. der zu verwaltende Wertetyp wird über die (Java 5) Generics-Funktionalität innerhalb der Implementierung festgelegt.



Des Weiteren muss die Methode `boolean likenTo(T value)` implementiert werden, die ermittelt, ob der Inhalt einer Eingabekomponente dem gegebenen Wert entspricht oder nicht.

Beispiel zur Verwendung des Aspekts siehe Kapitel 3.14.5.2 (Seite 135).

3.12.2.17 Aspekt: IntegrityValidating

Aspect: IntegrityValidating

Package: de.espirit.firstspirit.ui.gadgets.aspects

Sofern eine Eingabekomponente Werte eines bestimmten Typs speichern und bearbeiten kann (vgl. Aspekt: ValueHolder<T> in Kapitel 3.12.2.3 und Aspekt: Editable in Kapitel 3.12.2.4), kann es notwendig sein, die Datenintegrität des Persistenztyps zu prüfen. Mithilfe des Aspekts IntegrityValidating kann eine Validierung der Werte beim Speichern erfolgen.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements IntegrityValidating{
2.
3. ...
4.
5.     public mySwingGadget(final
        SwingGadgetContext<myGomForm> context) {
6.         super(context);
7.         addAspect(IntegrityValidating.TYPE, this);
8.     }
9.
10.     ...
11.
12. }
```



Des Weiteren muss die Methode `Set<? extends Problem> validateIntegrity()` implementiert werden, die prüft, ob die Eingabekomponente einen gültigen Persistenztyp enthält. Welche Art der Validierung ausgeführt wird, entscheidet dabei jede Komponente selbst. So kann beispielsweise eine Datums-Eingabekomponente den enthaltenen Wert parsen, um die Eingabe zu validieren:

```
1. @NotNull
2. public Set<? extends Problem> validateIntegrity() {
3.
4.     if (allowsInput()) {
5.         final String input = _component.getText();
6.         try {
7.             final DateFormat dateFormat = getDateFormat();
8.             dateFormat.parse(input);
9.         } catch (final ParseException e) {
10.            final DefaultProblem problem = new
                DefaultProblem("invalid date");
11.            problem.putContext("text", input);
12.            return Collections.singleton(problem);
13.        }
14.    }
15.    return Collections.emptySet();
16. }
```



3.12.2.18 Aspekt: TransferSupplying

Aspect: TransferSupplying

Package: de.espirit.firstspirit.ui.gadgets.aspects.transfer

Abhängig von der SwingGadget-Implementierung kann eine grafische Eingabekomponente (d.h. die Komponente muss `java.awt.Component` implementieren) einen generischen Zugriff auf ihre Inhalte bieten, um Objekte eines bestimmten Typs (Transfer-Typ) per Drag & Drop in andere Eingabekomponenten zu übernehmen. Dabei unterscheidet man zwei Aspekte, das Liefern und das Empfangen eines Objekts. Der Aspekt `TransferSupplying` ist für das Liefern eines Objekts per „Drag“ verantwortlich.

Dieser Aspekt kann der SwingGadget-Implementierung über die `Implements`-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements TransferSupplying {
2.
3. ...
4.
5.     public mySwingGadget (final
6.         SwingGadgetContext<myGomForm> context) {
7.         super (context);
8.         addAspect (TransferSupplying.TYPE, this);
9.     }
10.     ...
11.
12. }
```

Mithilfe der Methode `void registerSuppliers(@NotNull SupplierHost host)` muss die Komponente (oder ein Teil der Komponente, beispielsweise ein Eingabefeld) zunächst als Lieferant (Supplier) für die zu transferierenden Objekte registriert werden. Dazu werden vorab ein (oder mehrere) neue Supplier-Objekte erzeugt. Das Interface `Supplier<T>` stellt die Methode `List<T> supply()` für die Transferbehandlung zur Verfügung. Das Interface ist typisiert, d. h. der zu verwaltende Wertetyp (beispielsweise `Media`) wird über die (Java 5) Generics-Funktionalität innerhalb der Implementierung festgelegt.

Der Methode `void registerSuppliers(@NotNull SupplierHost host)` wird ein Objekt vom Typ `SupplierHost` übergeben, auf dem die erzeugten `Supplier` anschließend registriert werden müssen. Bei der Registrierung eines `Suppliers` muss ein Transfer-Typ übergeben werden. Der übergebene Transfer-Typ muss zum Daten-Container-Typ `T` des `Suppliers` passen.



Ein Transfer-Typ wird über einen `TransferAgent` erzeugt. Über den `SwingGadgetContext` (siehe Kapitel 3.12.5 Seite 96) kann eine Instanz vom Typ `SpecialistsBroker` (siehe Kapitel 3.12.4.1 Seite 95) angefordert werden. Ein `SpecialistsBroker` bietet über unterschiedliche „Spezialisten“ Zugriff auf bestimmte Dienste oder Informationen. Für das Erzeugen von Transfer-Typen, wird ein Spezialist vom Typ `TransferAgent` benötigt. Dieser kann auf dem `SpecialistBroker` mithilfe der Methode `<S> S requireSpecialist (SpecialistType<S> type)` angefordert werden. Bei der Registrierung eines `Suppliers` (auf dem `SupplierHost`) wird neben dem Transfer-Typ, noch das Ziel der Drag-Aktion und der entsprechende `Supplier` übergeben:

```
1. public void registerSuppliers(@NotNull final
   SupplierHost host) {
2.     final SupplierHost.Supplier<Media> supplier = new
       SupplierHost.Supplier<Media>() {
3.         public List<Media> supply() {
4.             final IDProvider element = _referenceField.getValue();
5.             return element instanceof Media ?
               Collections.singletonList((Media) element) :
               Collections.<Media>emptyList();
6.         }
7.     };
8.
9.     final TransferAgent transferAgent =
       getBroker().requireSpecialist(TransferAgent.TYPE);
10.
11.    final TransferType<Media> pictureType =
       transferAgent.getMediaType(Picture.class);
12.
13.    host.register(_preview, pictureType,
       supplier);
14.    host.register(_referenceField,
       pictureType, supplier);
15. }
```

Im Beispiel werden zwei Ziele, ein Eingabefeld (`_referenceField`) und ein Vorschaubereich (`_preview`), innerhalb der Eingabekomponente sowie ein Transfer-Typ für Objekte vom Typ `PICTURE` registriert. Damit können die enthaltenen Medien (ausschließlich Bilder) aus dem Eingabefeld oder dem Vorschaubereich der Eingabekomponente auf andere Komponenten gezogen werden („Drag“).

Die Drag- & Drop-Funktionalität einer Eingabekomponente kann ausschließlich über die Aspekte `TransferHandling` und `TransferSupplying` behandelt werden. Dazu müssen lediglich die dafür vorgesehenen Lieferanten und Empfänger registriert werden. Das umliegende FirstSpirit-Gadget-Framework behandelt dann alle weiteren Funktionen, beispielsweise das Drop-Handling. Beide Aspekte sind eigenständig, können also getrennt voneinander implementiert werden.



3.12.2.19 Aspekt: TransferHandling

Aspect: TransferHandling

Package: de.espirit.firstspirit.ui.gadgets.aspects.transfer

Abhängig von der SwingGadget-Implementierung kann eine grafische Eingabekomponente (d.h. die Komponente muss `java.awt.Component` implementieren) einen generischen Zugriff auf die Inhalte anderer Quellen erhalten, um Objekte eines bestimmten Typs (Transfer-Typ) per „Drop“ in die Eingabekomponente zu übernehmen. Dabei unterscheidet man zwei Aspekte, das Liefern und das Empfangen eines Objekts. Der Aspekt `TransferHandling` ist für das Empfangen eines Objekts per „Drop“ verantwortlich.

Dieser Aspekt kann der SwingGadget-Implementierung über die Implements-Bedingung und den Aufruf von `addAspect(...)` im öffentlichen Konstruktor der SwingGadget-Implementierung hinzugefügt werden:

```
1. public class mySwingGadget implements TransferHandling {
2. ...
3.     public mySwingGadget(final
           SwingGadgetContext<myGomForm> context) {
4.         super(context);
5.         addAspect(TransferHandling.TYPE, this);
6.     }
7. ...
8.
9. }
```

Mithilfe der Methode `void registerHandlers(@NotNull HandlerHost host)` muss die Komponente (oder ein Teil der Komponente, beispielsweise ein Eingabefeld) zunächst als Empfänger (Handler) für die zu transferierenden Objekte registriert werden. Dazu werden vorab ein (oder mehrere) neue Handler-Objekte erzeugt. Das Interface `Handler` stellt die Methode `void handle(@NotNull CommodityContainer commodities, final Point location)` für die Transferbehandlung zur Verfügung. Dieser Methode wird eine Instanz vom Typ `CommodityContainer` übergeben, die einen typsicheren Zugriff auf die zu übertragenden Objekte ermöglicht. Über den Aufruf der Methode `<A> List<A> get(TransferType<A> type)` auf der Instanz vom Typ `CommodityContainer` kann hier der gewünschte Transfer-Typ übergeben werden (siehe Beispiel Seite 93). Dabei kann ein `CommodityContainer` mehrere, unterschiedliche Transfer-Typen aufnehmen. Die dort übergebenen Transfer-Typen werden typisiert, d. h. der zu



verwaltende Wertetyp (beispielsweise `PICTURE`) wird über die (Java 5) Generics-Funktionalität innerhalb der Implementierung festgelegt.

```
1. public void registerHandlers(@NotNull final
   HandlerHost host) {
2.
3.     final HandlerHost.Handler imageHandler = new
   HandlerHost.Handler() {
4.         public void handle(@NotNull final CommodityContainer
   commodities, final Point location) throws
   TransferException {
5.             final TransferAgent transferAgent = _context.
   getBroker().requireSpecialist(TransferAgent.TYPE);
6.
7.             final List<Media> media = commodities.
   get(transferAgent.getMediaType(Picture.class));
8.             ...
9.         }
10.    };
11.
12.    final TransferAgent transferAgent = _context.
   getBroker().requireSpecialist(TransferAgent.TYPE);
13.
14.    final TransferType<Media> picType =
   transferAgent.getMediaType(Picture.class));
15.
16.    host.registerHandler(_preview, imageHandler,
   picType);
17.    host.registerHandler(_referenceField, imageHandler,
   picType);
18.
19.    final HandlerHost.Handler commentHandler = new
   HandlerHost.Handler() {
20.        public void handle(@NotNull final CommodityContainer
   commodities, final Point location) {
21.            final TransferAgent transferAgent = _context.
   getBroker().requireSpecialist(TransferAgent.TYPE);
22.
23.            final List<String> texts = commodities.get(
   transferAgent.getType("text/plain",
   String.class));
24.            ...
25.        }
26.    };
27.
28.    host.registerHandler(_commentField, commentHandler,
   transferAgent.getAllTextsType());
29.
30.    ...
31. }
```

Im Beispiel werden zwei Handler-Objekte erzeugt, die unterschiedliche Transfer-



Typen behandeln. Die Handler-Instanz `imageHandler` ist für Medien-Objekte vom Typ `Picture` zuständig, während die Handler-Instanz `commentHandler` Texte bzw. Objekte vom Typ `String` behandelt (siehe Beispiel auf Seite 93).

Ein Transfer-Typ wird über einen `TransferAgent` erzeugt. Über den `SwingGadgetContext` (siehe Kapitel 3.12.5 Seite 96) kann eine Instanz vom Typ `SpecialistsBroker` (siehe Kapitel 3.12.4.1 Seite 95) angefordert werden. Ein `SpecialistsBroker` bietet über unterschiedliche „Spezialisten“ Zugriff auf bestimmte Dienste oder Informationen. Für das Erzeugen von Transfer-Typen, wird ein Spezialist vom Typ `TransferAgent` benötigt. Dieser kann auf dem `SpecialistBroker` mithilfe der Methode `<S> S requireSpecialist (SpecialistType<S> type)` angefordert werden.

Der Methode `void registerHandlers(@NotNull HandlerHost host)` wird ein Objekt vom Typ `HandlerHost` übergeben, auf dem die erzeugten Handler anschließend registriert werden müssen. Dazu muss das Ziel der Drop-Aktion, die Menge der Transfer-Typen und der entsprechende Handler übergeben werden (siehe Beispiel auf Seite 93).

Im Beispiel werden drei Ziele, zwei Eingabefelder (`_referenceField` und `_commentField`) und ein Vorschaubereich (`_preview`) innerhalb der Eingabekomponente sowie die Transfer-Typen `PICTURE` und `STRING` registriert. Damit können Medien (ausschließlich vom Typ `Picture` und nicht vom Typ `File`) in das Eingabefeld oder den Vorschaubereich der Eingabekomponente gezogen und dort fallengelassen werden („Drop“). Desweiteren kann das zweite Eingabefeld (`_commentField`) einfache Texte bzw. Objekte vom Typ `String` aufnehmen.

Die Drag- & Drop-Funktionalität einer Eingabekomponente kann ausschließlich über die Aspekte `TransferHandling` und `TransferSupplying` behandelt werden. Dazu müssen lediglich die dafür vorgesehenen Lieferanten und Empfänger registriert werden. Das umliegende FirstSpirit-Gadget-Framework behandelt dann alle weiteren Funktionen, beispielsweise das Drop-Handling. Beide Aspekte sind eigenständig, können also getrennt voneinander implementiert werden.



3.12.3 Agents

Die Dokumentation in diesem Bereich ist noch nicht vollständig abgeschlossen.

3.12.3.1 OperationAgent

Verweis auf Kapitel 3.12.8.5 Seite 115.

3.12.3.2 TransferAgent

Verweis auf den SwingGadget-Aspekt `TransferHandling` in Kapitel 3.12.2.19 (Seite 92).

3.12.4 Broker

Ein Broker bietet dem Komponentenentwickler Zugriff auf auf bestimmte Dienste oder Informationen.

3.12.4.1 SpecialistBroker

Die Dokumentation in diesem Bereich ist noch nicht vollständig abgeschlossen.

Im Zuge der Refaktorisierung des FirstSpirit-Komponentenmodells war es notwendig, die Abhängigkeit der Werteträger vom `UserService` auszubauen und durch den `SpecialistBroker` zu ersetzen.

Über den `SwingGadgetContext` kann eine Instanz vom Typ `SpecialistsBroker` angefordert werden. Ein `SpecialistsBroker` bietet über unterschiedliche „Spezialisten“ Zugriff auf bestimmte Dienste oder Informationen. Für das Erzeugen von Referenzen, wird ein Spezialist vom Typ `ReferenceConstructionAgent` benötigt. Dieser kann auf dem `SpecialistBroker` mithilfe der Methode `<S> S requireSpecialist(SpecialistType<S> type)` angefordert werden:

3.12.4.2 ServiceBroker

Die Dokumentation in diesem Bereich ist noch nicht vollständig abgeschlossen.



3.12.5 SwingGadgetContext

Der SwingGadgetContext wird bei der Erzeugung eines neuen spezifischen Swing-Gadget-Editors übergeben. Er enthält wichtige Informationen, die für die Implementierung des Editors benötigt werden. Beispielsweise kann über den SwingGadgetContext der *NotificationHost* angesprochen werden, der Methoden für die Bekanntgabe von Ereignissen bzw. Änderungen, die innerhalb des Editors stattfinden, zur Verfügung stellt.

Die Dokumentation in diesem Bereich ist noch nicht vollständig abgeschlossen. Ein Beispiel zum Arbeiten mit SwingGadgets ist in Kapitel 3.14.3 (Seite 129) beschrieben



3.12.6 Wertespeicherung (EditorValue, ValueEngineer)

3.12.6.1 Das Interface EditorValue

Package: `de.espirit.firstspirit.access.editor`

Abhängig von der SwingGadget-Implementierung kann eine Eingabekomponente die Fähigkeit haben, Werte zu speichern. (vgl. Aspekt: ValueHolder<T> in Kapitel 3.12.2.3). Alle FirstSpirit-Eingabekomponenten speichern ihre Werte in sogenannten EditorValues. Das Interface `EditorValue<T>` ist sehr komplex und beinhaltet umfangreiche Methoden für den Zugriff auf EditorValues, die zum größten Teil für die Entwicklung kundenspezifischer Eingabekomponenten nicht benötigt werden. Bisher wurde daher die Erweiterung der EditorValue-Implementierung mit der abstrakten Basisimplementierung `AbstractEditorValue<T>` empfohlen, die grundlegende Funktionalität, wie das Parsen oder Formatieren von Werten eines SwingGadgets bzw. für deren XML-Repräsentation in der jeweiligen Sprache zur Verfügung stellt. `AbstractEditorValue<T>` ist jedoch kein Bestandteil der öffentlichen FirstSpirit-API, erfüllt also auch nicht deren Stabilitätsanforderungen, was in der Vergangenheit zu Kompatibilitätsproblemen führte.

Um die Entwicklung im Bereich der EditorValues zu vereinfachen und eine stabile Entwicklungsumgebung zu gewährleisten, werden mit FirstSpirit Version 4.2R4 neue Interfaces eingeführt, die in den nachfolgenden Kapiteln beschrieben werden. Das Interface EditorValue wird ab FirstSpirit Version 4.2R4 nur noch intern verwendet.

3.12.6.2 Das Interface ValueEngineerFactory

Package: `de.espirit.firstspirit.client.access.editor`

Das Interface `ValueEngineerFactory<T, F extends GomFormElement>` ist eine Factory, die zur Erzeugung eines neuen typisierten Objekts vom Typ `ValueEngineer<T>` dient (siehe Kapitel 3.12.6.3 Seite 98).

Das Interface ist typisiert, d. h. der zu verwaltende Wertetyp (Persistenztyp) und das zugehörige Formular-Element werden über die (Java 5) Generics-Funktionalität innerhalb der einzelnen Implementierungen festgelegt.

- `Class<T> getType()`:
Damit das SwingGadget mit dem korrekten Persistenztyp des zugehörigen EditorValues arbeitet, ist innerhalb der ValueEngineerFactory-Implementierung zunächst die Angabe des Persistenztyps erforderlich. Dazu muss die Methode



`Class<T> getType()` implementiert werden. Die Methode liefert die Klasse zurück, die den Persistenztyp der Eingabekomponente repräsentiert, wobei der Parameter `Typ T` zum Daten-Container-Typ des `EditorValue<T>` (siehe Kapitel 3.12.6.1 Seite 97) und zum Wertetyp der zugehörigen `SwingGadget`-Implementierung (siehe Kapitel 3.12.2.3 Seite 73) passen muss (siehe auch „Wertetypen“ in Kapitel 3.12.6.5 (Seite 102)).

- `ValueEngineer<T> create(ValueEngineerContext<F> context):` Die Methode dient zur Erzeugung einer neuen Instanz vom Typ `ValueEngineer<T>` (siehe Kapitel 3.12.6.3 Seite 98). Der Methode wird ein typisierter `ValueEngineerContext<F>` übergeben, wobei der Parameter `F` den Typ des zugehörigen Formular-Elements (GOMForm) bezeichnet (siehe Kapitel 3.12.6.4 Seite 101). Der Kontext enthält weitere Informationen (Formular, Sprache, Release-Flag, `SpecialistBroker`), die abhängig vom Funktionsumfang der Eingabekomponente, bei der weiteren Implementierung benötigt werden (beispielsweise zur Erzeugung von Referenzen). Die Factory verknüpft so die Implementierung der Persistenz einer Eingabekomponente mit dem zugrundeliegenden Formular-Element (GOM-Form).

3.12.6.3 Das Interface `ValueEngineer<T>`

Package: `de.espirit.firstspirit.client.access.editor`

Um die Entwicklung im Bereich der `EditorValues` zu vereinfachen und eine stabile Entwicklungsumgebung zu gewährleisten (vgl. Kapitel 3.12.6.1), wird mit FirstSpirit Version 4.2R4 das neue Interface `ValueEngineer<T>` extends `Aspectable` eingeführt. Das Interface `ValueEngineer<T>` stellt Basisfunktionalitäten bereit, die für die Implementierung kundenspezifischer Eingabekomponenten benötigt werden, hauptsächlich das Lesen und Schreiben des Persistenzwertes einer Eingabekomponente. Es bildet damit eine übersichtliche und stabile Schnittstelle zum Interface `EditorValue<T>` (siehe Kapitel 3.12.6.1 Seite 97), welches ab FirstSpirit Version 4.2R4 nur noch intern verwendet wird.

Das Interface `ValueEngineer<T>` erbt vom Interface `Aspectable` und unterstützt die Methode `<T> T getAspect(@NotNull AspectType<T> aspect)`. Das heißt, neue Funktionalität, die über die Basisfunktion des Interfaces `ValueEngineer<T>` hinausgeht, muss nicht von der `ValueEngineer`-Implementierung selber umgesetzt werden, sondern kann als Aspekt angefragt werden (siehe Kapitel 3.12.7 Seite 102). Die bisherige Implementierung bleibt kompatibel und muss nicht angepasst werden.



Das Interface `ValueEngineer<T>` ist typisiert, d. h. der zu verwaltende Wertetyp (Persistenztyp) wird über die (Java 5) Generics-Funktionalität innerhalb der einzelnen Implementierungen festgelegt. Der zurückgelieferte Persistenztyp muss zum Daten-Container-Typ des `EditorValue<T>` und zu den Wertetypen der zugehörigen `SwingGadget`- und `ValueEngineerFactory`-Implementierungen passen (siehe auch „Wertetypen“ in Kapitel 3.12.6.5 (Seite 102)).

Es müssen die folgenden Methoden implementiert werden:

- `T read(@NotNull List<Node> nodes):`
Die Methode ist zuständig für das Laden der `EditorValues` (Xml to Object). Dazu wird eine Liste von Knoten (Nodes) übergeben. Ein Objekt vom Typ `Node` ist ein Daten-Container, der einen Namen, Attribute (optional) und entweder einen textuellen Wert oder weitere Objekte vom Typ `Node` (Kind-Knoten) enthält. Die für jeden Knoten gespeicherten Werte werden als `String` zurückgeliefert und müssen geeignet konvertiert werden. Über die Methode `void setValue(@Nullable T value)` aus dem Interface `ValueHolder<T>` (vgl. Kapitel 3.12.2.3) wird dieser Wert typsicher an das `SwingGadget` geliefert. Sind die übergebenen Werte sprachabhängig, muss die korrekte Zielsprache auch bei der weiteren Implementierung (basierend auf dem Rückgabewert) berücksichtigt werden. Die Zielsprache kann über den über den Aufruf von `ValueEngineerContext.getLanguage()` ermittelt werden (siehe Kapitel 3.12.6.4 Seite 101).
- `List<Node> write(@NotNull T value):`
Über die Methode `T getValue()` aus dem Interface `ValueHolder<T>` (vgl. Kapitel 3.12.2.3) werden die Werte aus dem `SwingGadget` gelesen. Der zurückgelieferte Wert vom Typ `T` wird an die Methode `List<Node> write(@NotNull T value)` übergeben. Die Methode konvertiert den übergebenen Wert in ein (oder mehrere) Objekte vom Typ `Node` und liefern diese innerhalb einer Liste von Knoten (Nodes) zurück (Object to Xml). Ein `Node` kann neben dem textuellen Wert (oder weiteren Objekten vom Typ `Node`) noch einen Namen, der der Spezifikation entsprechen muss, und optionale Attribute enthalten. Die Methode `List<Node> write(@NotNull T value)` wird bei jedem Speichervorgang im `JavaClient` aufgerufen.
- `T getEmpty():` Die Methode erzeugt eine neue, leere Instanz eines Persistenztyps (Leerwert). Dieser wird insbesondere bei komplexen Werten benötigt, beispielsweise um eine leere Liste zu erzeugen. Für einfache Persistenztypen, beispielsweise `String`, kann null zurückgeliefert werden (siehe auch „Wertetypen“ in Kapitel 3.12.6.5 (Seite 102)).



- `boolean isEmpty(@NotNull T value)`: Komplexe Wertetypen sind nie null, möglicherweise jedoch leer (s.o.). Diese Methode prüft, ob die Instanz eines komplexen Persistenztyps, beispielsweise eine Liste, leer ist oder nicht.
- `T copy(@NotNull T original)`: Die Methode erzeugt eine neue Instanz des übergebenen Persistenztyps mit identischen Inhalten (Kopie). Für einfache, unveränderliche Datentypen (beispielsweise Strings), kann an dieser Stelle auch die Original-Instanz zurückgeliefert werden, für alle komplexeren Wertetypen sollte aber darauf geachtet werden, dass eine Änderung der Original-Instanz keine Auswirkung auf die Kopie hat:

```
original != copy
```

Ist der übergebene Wertetyp sprachabhängig, muss die korrekte Zielsprache auch bei der weiteren Implementierung (basierend auf dem Rückgabewert) berücksichtigt werden. Die Zielsprache kann über den über den Aufruf von `ValueEngineerContext.getLanguage()` ermittelt werden (siehe Kapitel 3.12.6.4 Seite 101).

- `<T> T getAspect(@NotNull ValueEngineerAspectType<T> aspect)`
Das Interface `ValueEngineer<T>` erbt vom Interface `Aspectable` und unterstützt die Methode `<T> T getAspect(@NotNull AspectType<T> aspect)`. Neue Funktionalität, die über die Basisfunktion des Interfaces `ValueEngineer<T>` hinausgeht, muss nicht von der `ValueEngineer`-Implementierung selber umgesetzt werden, sondern steht als Interface (Aspekt-Typ) zur Verfügung. Die `ValueEngineer`-Implementierung muss nur noch die entsprechenden Methoden des Interfaces implementieren, das umliegende FirstSpirit-Framework behandelt dann automatisch alle weiteren Funktionen. Die Methode `<T> T getAspect(@NotNull AspectType<T> aspect)` erzeugt eine aspektorientierte Instanz des Persistenztyps. Die unterschiedlichen Aspekt-Typen stellen spezielle Anforderungen an die `ValueEngineer`-Implementierung, die in Kapitel 3.12.7 (Seite 102 ff.) erläutert werden.

Ein Beispiel für die Verwendung eines Aspekts innerhalb der ValueEngineer-Implementierung befindet sich in Kapitel 3.14.7 Seite 137.



3.12.6.4 Das Interface ValueEngineerContext<F extends GomFormElement>

Package: `de.espirit.firstspirit.client.access.editor`

Der typisierte `ValueEngineerContext<F extends GomFormElement>` wird Objekten vom Typ `ValueEngineer<T>` während der Erzeugung übergeben, wobei der Parameter `F` den Typ des zugehörigen Formular-Elements (GOMForm) bezeichnet. Der Kontext enthält weitere Informationen (Formular, Sprache, Release-Flag, `SpecialistBroker`), die abhängig vom Funktionsumfang der Eingabekomponente, bei der weiteren Implementierung benötigt werden. Das Release-Flag, das anzeigt, ob der Wert einer Eingabekomponente freigegeben ist (oder nicht) wird beispielsweise bei der Erzeugung von Referenzen benötigt (siehe Kapitel 3.12.8 Seite 109).

Das Interface bietet den Zugriff auf folgende Methoden:

- `F getGom()` :
Liefert das Formular-Element der zugehörigen Eingabekomponente (siehe Kapitel 3.14.2 Seite 127).
- `SpecialistsBroker getBroker()` :
Liefert ein Objekt vom Typ `SpecialistBroker` zurück, das weitere Informationen, beispielsweise bestimmte, zur Eingabekomponente gehörige `StoreElemente` liefern kann (siehe Kapitel 3.12.4.1 Seite 95) .
- `Language getLanguage()` :
Eine Eingabekomponente kann ihre Werte sprachabhängig oder sprachunabhängig speichern. Die Methode liefert die gewünschte Zielsprache für die Wertespeicherung zurück.
- `boolean isRelease()` :
Die Werte einer Eingabekomponente können über einen Arbeitsablauf für den Veröffentlichungsprozess freigegeben werden (Release). Die Methode liefert zurück, ob der `ValueEngineer` auf dem Freigabe-Stand arbeitet oder nicht.



3.12.6.5 Wertetypen

Persistenztypen teilen sich in komplexe, kapselnde und einfachen Werte auf:

- Komplexe Werte: Sind natürliche Container für die referenzierten Werte. Änderungen innerhalb der komplexen Werte haben direkte Auswirkung auf den Wert der Eingabekomponente. Komplexe Werte sind nie null, möglicherweise jedoch leer. Zu den komplexen Werten gehören Listen und Mengen.
- Kapselnde Werte: Sind einfache Werte, die einen "inneren" Wert auf sich selbst abbilden. Die Änderung ihres Wertes hat direkte Auswirkung auf den Wert der Eingabekomponente. Kapselnde Werte können null sein und müssen in diesem Fall explizit gesetzt werden. Zu den kapselnden Werten gehören: Date, Link
- Einfache Werte: Sind unveränderbare Werte, die direkt den eigentlichen Wert darstellen. Änderungen können ausschließlich durch Setzen des Wertes in der Eingabekomponente erfolgen. Zu den einfachen Werten gehören: String, Number.

3.12.7 Aspekte (ValueEngineer)

Um die Stabilität des Interfaces `ValueEngineer<T>` zu gewährleisten, wird alle Funktionalität, die über die bisher implementierte Basisfunktionalität hinausgeht (vgl. Kapitel 3.12.6.3 Seite 98), über Aspekte realisiert. Die unterschiedlichen ValueEngineer-Aspekt-Typen stehen als Interface zur Verfügung. Die ValueEngineer-Implementierung muss die gewünschten Aspekte lediglich in der Methode `<T> T getAspect(@NotNull ValueEngineerAspectType<T> aspect)` bekanntgeben und die entsprechenden Methoden des Interfaces implementieren. Das umliegende FirstSpirit-Framework behandelt dann automatisch alle weiteren Funktionen.

Aspekte können auch innerhalb der SwingGadget-Implementierung eingesetzt werden (siehe Aspekte (SwingGadget) in Kapitel 3.12.2 Seite 69). Die in diesem Kapitel beschriebenen Aspekte (vom Typ `ValueEngineerAspectType`) beziehen sich immer auf den Wert eines SwingGadgets, nicht auf das SwingGadget selbst. ValueEngineer- und SwingGadget-Aspekte können aber eng miteinander verknüpft sein. Soll beispielsweise eine Suchtreffer-Markierung realisiert werden, so muss dazu sowohl der SwingGadget-Aspekt `Highlightable` als auch der ValueEngineer-Aspekt `MatchSupporting` implementiert werden (Beispiel siehe Kapitel 3.14.9 Seite 142).



Bislang sind folgende ValueEngineer-Aspekte verfügbar:

- `Aspekt: MatchSupporting: Aufbereiten der im ValueEngineer enthaltenen Werte für die Indizierung und Suche` (siehe Kapitel 3.12.7.1 Seite 104) Der Aspekt ist eng verknüpft mit dem `SwingGadget-Aspekt Highlightable` (siehe Kapitel 3.12.2.13 Seite 83).
- `Aspekt: DifferenceComputing: Aufbereiten der im ValueEngineer enthaltenen Werte für die Differenz-Visualisierung` (siehe Kapitel 3.12.7.2 Seite 105). Der Aspekt ist eng verknüpft mit dem `SwingGadget-Aspekt DifferenceVisualising` (siehe Kapitel 3.12.2.14 Seite 84).
- `Aspekt: ReferenceContaining: Erstellen von ausgehenden Referenzen für die im ValueEngineer enthaltenen Werte` (siehe Kapitel 3.12.7.3 Seite 107).



3.12.7.1 Aspekt: MatchSupporting

Aspect:	MatchSupporting
Package:	de.espirit.firstspirit.client.access.editor
<p>Die innerhalb eines ValueEngineers enthaltenen Werte können durchsucht werden. Diese Funktionalität wird über den Aspekt MatchSupporting bereitgestellt. Die im ValueEngineer enthaltenen Werte müssen zunächst für eine Suche aufbereitet und indiziert werden. Nur die entsprechenden Methoden des Interfaces MatchSupporting müssen vom Komponentenentwickler implementiert werden, das umliegende FirstSpirit-Framework behandelt automatisch alle weiteren Funktionen, beispielsweise die Indizierung.</p> <p>Dieser Aspekt kann der ValueEngineer-Implementierung über die Methode <code><T> T getAspect(@NotNull ValueEngineerAspectType<T> aspect)</code> aus dem Interface <code>ValueEngineer<T></code> hinzugefügt werden (siehe Kapitel 3.12.6.3 Seite 98). Die Methode liefert eine aspektorientierte Instanz des Persistenztyps zurück:</p>	
<pre> 1. public <T> T getAspect(@NotNull final ValueEngineerAspectType<T> aspect) { 2. if (aspect == MatchSupporting.TYPE) { 3. return aspect.cast(this); 4. } 5. return null; 6. }</pre>	
<p>Außerdem müssen die Methoden <code>String getStringForIndex(@NotNull T value)</code> und <code>List<? extends Match> getMatches(@NotNull Request request, @NotNull T value)</code> implementiert werden. Die erste Methode konvertiert den übergebenen Wert in einen String, damit die Inhalte für die Suche indiziert werden können. Die zweite Methode liefert eine Liste von Treffern, die einem übergebenen Suchmuster entsprechen. Diese Liste wird beispielsweise zur Visualisierung der Suchtreffer an die SwingGadget-Implementierung weitergeleitet.</p> <p><i>Beispiel zur Implementierung des SwingGadget-Aspekts Highlightable siehe Kapitel 3.14.9 und zur Implementierung des ValueEngineer-Aspekts MatchSupporting siehe Kapitel 3.14.7.</i></p>	



3.12.7.2 Aspekt: DifferenceComputing

Aspect: DifferenceComputing

Package: de.espirit.firstspirit.client.access.editor

Innerhalb der ValueEngineer-Implementierung kann der Unterschied zwischen zwei gegebenen Werten ermittelt werden. Diese Funktionalität wird über den Aspekt `DifferenceComputing` bereitgestellt.

Dieser Aspekt kann der ValueEngineer-Implementierung über die Methode `<T> T getAspect(@NotNull ValueEngineerAspectType<T> aspect)` aus dem Interface `ValueEngineer<T>` hinzugefügt werden (siehe Kapitel 3.12.6.3 Seite 98). Die Methode liefert eine aspektorientierte Instanz des Persistenztyps zurück:

```
1. public <T> T getAspect(@NotNull final
   ValueEngineerAspectType<T> aspect) {
2.     if (aspect == DifferenceComputing.TYPE) {
3.         return aspect.cast(this);
4.     }
5.     return null;
6. }
```

Außerdem muss die Methode `List<Difference> computeDifferences(@NotNull T actualValue, @NotNull T oldValue)` implementiert werden, die eine Liste von Differenzen (`List<Difference>`) zwischen zwei übergebenen Werten liefern soll. Ein Objekt vom Typ `Difference` ist ein Daten-Container, der eine Beschreibung der Differenz in Form des geänderten Inhalts und einer zugehörigen Änderungsart (`Modification`) enthält. Man unterscheidet die Änderungsarten `NONE`, `INSERTED`, `DELETED` und `CHANGED`. Jede Änderung wird vom FirstSpirit-Framework unterschiedlich dargestellt.

Man unterscheidet vier Fälle:

- 1) Einer der beiden übergebenen Werte ist `null`: In diesem Fall wird vom FirstSpirit-Framework automatisch eine Liste mit einem einzelnen Differenzeintrag zurückgeliefert, der den gesamten aktuellen Wert als „CHANGED“ beschreibt. Die Methode `List<Difference> computeDifferences(@NotNull T actualValue, @NotNull T oldValue)` wird nicht aufgerufen.



- 2) Beide übergebene Werte sind `null`: In diesem Fall wird vom FirstSpirit-Framework automatisch eine Liste mit einem einzelnen Differenzeintrag zurückgeliefert, der den Inhalt als „nicht verändert“ beschreibt (`Modification.NONE`). Die Methode `List<Difference> computeDifferences(@NotNull T actualValue, @NotNull T oldValue)` wird nicht aufgerufen.
- 3) Beide Werte sind identisch, es liegen also keine Unterschiede vor: In diesem Fall kann der Komponentenentwickler eine leere Liste zurückliefern.
- 4) Beide Werte unterscheiden sich, beispielsweise zwei Strings mit folgendem Inhalt:

```
actualValue == "This is the actual value!",  
oldValue == "This is the old value."
```

In diesem Fall gibt es zwei Lösungsmöglichkeiten:

- a. Allgemeine Beschreibung (geringer Implementierungsaufwand):
Der Komponentenentwickler liefert eine Liste mit einem einzelnen Eintrag zurück, der den gesamten aktuellen Wert als „CHANGED“ beschreibt:
- b. Detaillierte Beschreibung (insbesondere bei textuellen Vergleichen mit hohem Implementierungsaufwand verbunden): Jeder Unterschied wird einzeln herausgefiltert und mit einer bestimmten Änderungsart als Differenzbeschreibung zur Liste hinzugefügt:

```
return Arrays.asList(  
    new Difference("This is the ", Modification.NONE),  
    new Difference("actual", Modification.INSERTED),  
    new Difference("old", Modification.DELETED),  
    new Difference(" value", Modification.NONE),  
    new Difference("!", Modification.INSERTED),  
    new Difference(".", Modification.DELETED));
```

Die zurückgelieferte Liste kann beispielsweise zur Visualisierung der Differenzen an die `SwingGadget`-Implementierung weitergeleitet werden (siehe Kapitel 3.12.2.14 Seite 84).



3.12.7.3 Aspekt: ReferenceContaining

Aspect: ReferenceContaining<T>
Package: de.espirit.firstspirit.client.access.editor

Wesentliche Funktionalitäten von FirstSpirit basieren auf dem so genannten Referenzgraph eines Projekts. Der Referenzgraph wird verwendet, um die Abhängigkeiten von Objekten innerhalb komplexer Projekte zu erkennen. Mithilfe des Aspekts `ReferenceContaining` können Komponententwickler nun erstmalig ausgehende Referenzen (References) für Eingabekomponenten implementieren (siehe Kapitel 3.12.8 Seite 109).

Dieser Aspekt kann der ValueEngineer-Implementierung über die Methode `<T> T getAspect(@NotNull ValueEngineerAspectType<T> aspect)` aus dem Interface `ValueEngineer<T>` hinzugefügt werden (siehe Kapitel 3.12.6.3 Seite 98). Die Methode liefert eine aspektorientierte Instanz des Persistenztyps zurück:

```
1. public <T> T getAspect(@NotNull final ValueEngineerAspectType<T>
   aspect) {
2.     if (aspect == ReferenceContaining.TYPE) {
3.         return aspect.cast(this);
4.     }
5.     return null;
6. }
```

Außerdem muss die Methode `List<Reference> collectReferences(@NotNull T value)` implementiert werden, die alle ausgehenden Referenzen des übergebenen Wertes als Liste zurückliefert. Die erforderlichen Referenzen müssen über die ValueEngineer-Implementierung erzeugt werden. Dazu wird eine Instanz vom Typ `ReferenceConstructionAgent` benötigt. Ein Objekt vom Typ `ReferenceConstructionAgent` kann über einen `SpecialistsBroker` angefordert werden. Der `SpecialistsBroker` steht im `ValueEngineerContext` zur Verfügung. Um innerhalb der ValueEngineer-Implementierung Zugriff auf den `ValueEngineerContext` zu haben, muss dieser bei der Erzeugung der ValueEngineer-Instanz innerhalb der ValueEngineerFactory-Implementierung weitergereicht werden (Beschreibung siehe Kapitel 3.12.8.5 Seite 115).



Über den `ReferenceConstructionAgent` können Instanzen vom Typ `ReferenceHolder` erzeugt werden (siehe Kapitel 3.12.8.4 Seite 113). Der `ReferenceHolder` hält die Informationen über das (interne oder externe) Objekt, das bei der Erzeugung der Instanz übergeben wurde. Die eigentliche Referenz wird erst durch den Aufruf der Methode `toReference(...)` auf der Instanz vom Typ `ReferenceHolder` erzeugt. Die Referenz kann schließlich innerhalb einer Liste zurückgeliefert werden.

```
1. @NotNull
2. public List<Reference> collectReferences(@NotNull final String
   value) {
3.     final SpecialistsBroker broker = _context.getBroker();
4.     final ReferenceConstructionAgent agent =
       broker.requireSpecialist(ReferenceConstructionAgent.TYPE);
5.     final ExternalReferenceHolder referenceHolder =
       agent.create(value, "custom");
6.     final Reference reference =
       referenceHolder.toReference(null);
7.
8.     return Collections.singletonList(reference);
9. }
```

Die Liste der Referenzen wird vom FirstSpirit-Framework an den Referenzmanager weitergereicht. Alle eingehenden und ausgehenden Referenzen eines `SwingGadgets` (bzw. der dort enthaltenen Werte) werden vom FirstSpirit Referenzmanager gesammelt und können über den Referenzgraph im FirstSpirit-JavaClient visuell dargestellt werden, beispielsweise über den Menüeintrag „Suche – externe Referenzen“.

Eine umfangreiche Beschreibung zum Arbeiten mit Referenzen befindet sich in Kapitel 3.12.8 Seite 109.



3.12.8 Arbeiten mit Referenzen

3.12.8.1 Übersicht

Wesentliche Funktionalitäten von FirstSpirit basieren auf dem Referenzgraph eines Projekts. Der Referenzgraph wird verwendet, um die Abhängigkeiten von Objekten innerhalb komplexer Projekte zu erkennen. Abhängigkeiten eines Objekts können in Form von eingehenden und ausgehenden Kanten sowohl für den aktuellen Stand als auch den zuletzt freigegebenen Stand (Release-Stand) beschrieben werden. Dabei entscheidet prinzipiell jede Eingabekomponente selber, auf welche Objekte sie referenziert.

Mithilfe des Aspekts `ReferenceContaining` (siehe Kapitel 3.12.7.3 Seite 107) können Komponententwickler nun erstmalig selber Referenzen für Eingabekomponenten erstellen. Der Aspekt muss innerhalb der ValueEngineer-Implementierung hinzugefügt werden, da der Wert einer Eingabekomponente die Referenz hält und nicht die Eingabekomponente (`SwingGadget`) selbst. Ein `SwingGadget` ist lediglich die grafische Repräsentation einer Eingabekomponente in der FirstSpirit-Redaktionsumgebung (siehe Kapitel 3.12.1 Seite 69), der FirstSpirit Referenzmanager referenziert auf dem FirstSpirit-Server nur die im `SwingGadget` gespeicherten Werte. Zur Erzeugung einer Referenz müssen die ausgehenden Kanten aus den vorhandenen Werten extrahiert und in einem Daten-Container vom Typ `ReferenceHolder` gespeichert werden (siehe Kapitel 3.12.8.4). Eine Übersicht über das Erstellen und Auslesen einer Referenz aus einem gegebenen Wert bieten die Kapitel 3.12.8.2 (Erstellen einer Referenz) und 3.12.8.3 (Auslesen einer Referenz). Die beteiligten Schnittstellen werden ab Kapitel 3.12.8.4 erläutert.

3.12.8.2 Erstellen einer Referenz (Write)

Das Erstellen bzw. Schreiben einer neuen Referenz läuft in folgenden Schritten ab:

- 1) Anfordern eines `ReferenceConstructionAgents` über einen `SpecialistsBroker` (Beschreibung und Beispiel siehe Kapitel 3.12.8.5 Seite 115).
- 2) Erzeugen einer neuen Instanz vom Typ `ReferenceHolder` für die Aufnahme einer internen oder externen Referenz durch den Aufruf der `create`-Methode auf der Instanz vom Typ `ReferenceConstructionAgents` (Beschreibung und Beispiel siehe Kapitel 3.12.8.5 Seite 115) (Zum Interface `ReferenceHolder` siehe Kapitel 3.12.8.4 Seite 113).



- 3) Anfordern eines `ReferenceTransformationAgents` über einen `SpecialistsBroker` (siehe Kapitel 3.12.8.6 Seite 117).
- 4) Aufruf der `write`-Methode auf der Instanz vom Typ `ReferenceTransformationAgent` und Übergabe des erzeugten `ReferenceHolder`s. Die Methode ist zuständig für die Übernahme der Werte aus der Instanz vom Typ `ReferenceHolder` in ihre Persistenzform (Object to Xml). Die Methode konvertiert den übergebenen `ReferenceHolder` dazu in ein Objekt vom Typ `Node`.
- 5) Die Methode liefert die Instanz vom Typ `Node` an die `write`-Methode des Interfaces `ValueEngineer` zurück. Diese liefert den Knoten (und optional weitere Knoten) innerhalb einer Liste von Knoten (Nodes) zurück (siehe Kapitel 3.12.6.3 Seite 98).

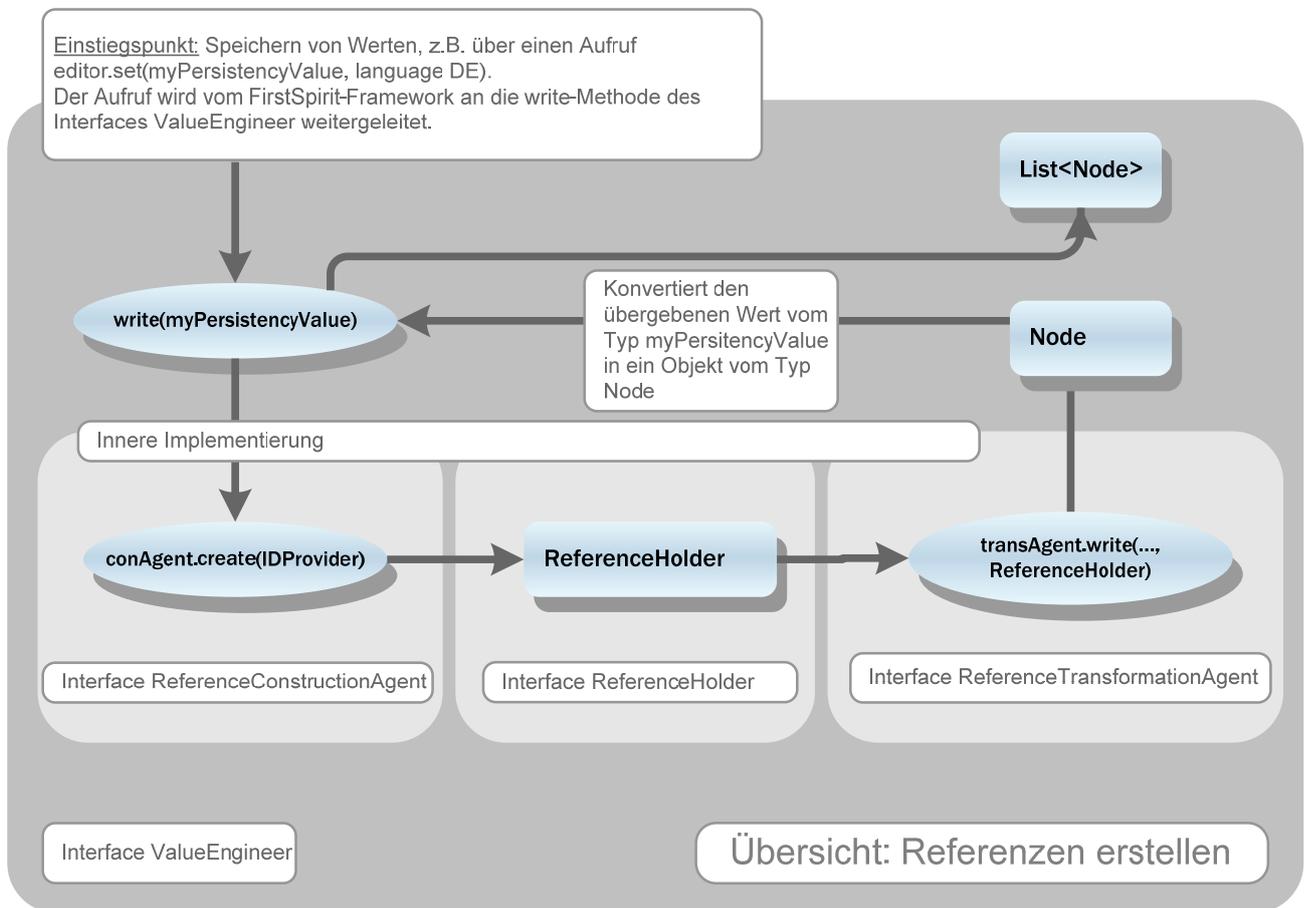


Abbildung 3-2: Schreiben von Referenzen



3.12.8.3 Auslesen einer Referenz (Read)

Das Auslesen einer neuen Referenz läuft in folgenden Schritten ab:

- 1) Die `read`-Methode des Interfaces `ValueEngineer` ist zuständig für das Laden der `EditorValues` (Xml to Object). Der Methode wird dazu eine Liste von Knoten (`Nodes`) übergeben. Die für jeden Knoten gespeicherten Werte werden als Instanz des gewünschten Persistenztyps zurückgeliefert und müssen innerhalb der inneren Implementierung geeignet konvertiert werden (siehe Kapitel 3.12.6.3 Seite 98).
- 2) Anfordern eines `ReferenceTransformationAgents` über einen `SpecialistsBroker` (siehe Kapitel 3.12.8.6 Seite 117).
- 3) Erzeugen einer neuen Instanz vom Typ `ReferenceHolder` für die Aufnahme einer internen oder externen Referenz durch den Aufruf der `read`-Methode auf der Instanz vom Typ `ReferenceConstructionAgents`. Die `read`-Methode ist zuständig für das Lesen der Werte aus der Persistenz. Dazu wird die Instanz vom Typ `Node` übergeben und in eine Instanz vom Typ `ReferenceHolder` überführt (Xml to Object) (Beschreibung und Beispiel siehe Kapitel 3.12.8.5 Seite 115) (Zum Interface `ReferenceHolder` siehe Kapitel 3.12.8.4 Seite 113).
- 4) Erzeugen einer neuen Referenz (bzw. einer neuen Instanz vom Typ `Reference`) durch den Aufruf der Methode `toReference(...)` auf dem `ReferenceHolder` (Beschreibung und Beispiel siehe Kapitel 3.12.8.4 Seite 113) (Zum Interface `Reference` siehe Kapitel 3.12.8.7 Seite 119).
- 5) Erzeugen einer neuen Instanz des gewünschten Persistenztyps durch den Aufruf der Methode `getReferencedObject()` auf der Instanz vom Typ `Reference`. (siehe Kapitel 3.12.8.6 Seite 117). Die Instanz des Persistenztyps wird an die `read`-Methode des Interfaces `ValueEngineer` zurückgeliefert (siehe Kapitel 3.12.6.3 Seite 98).



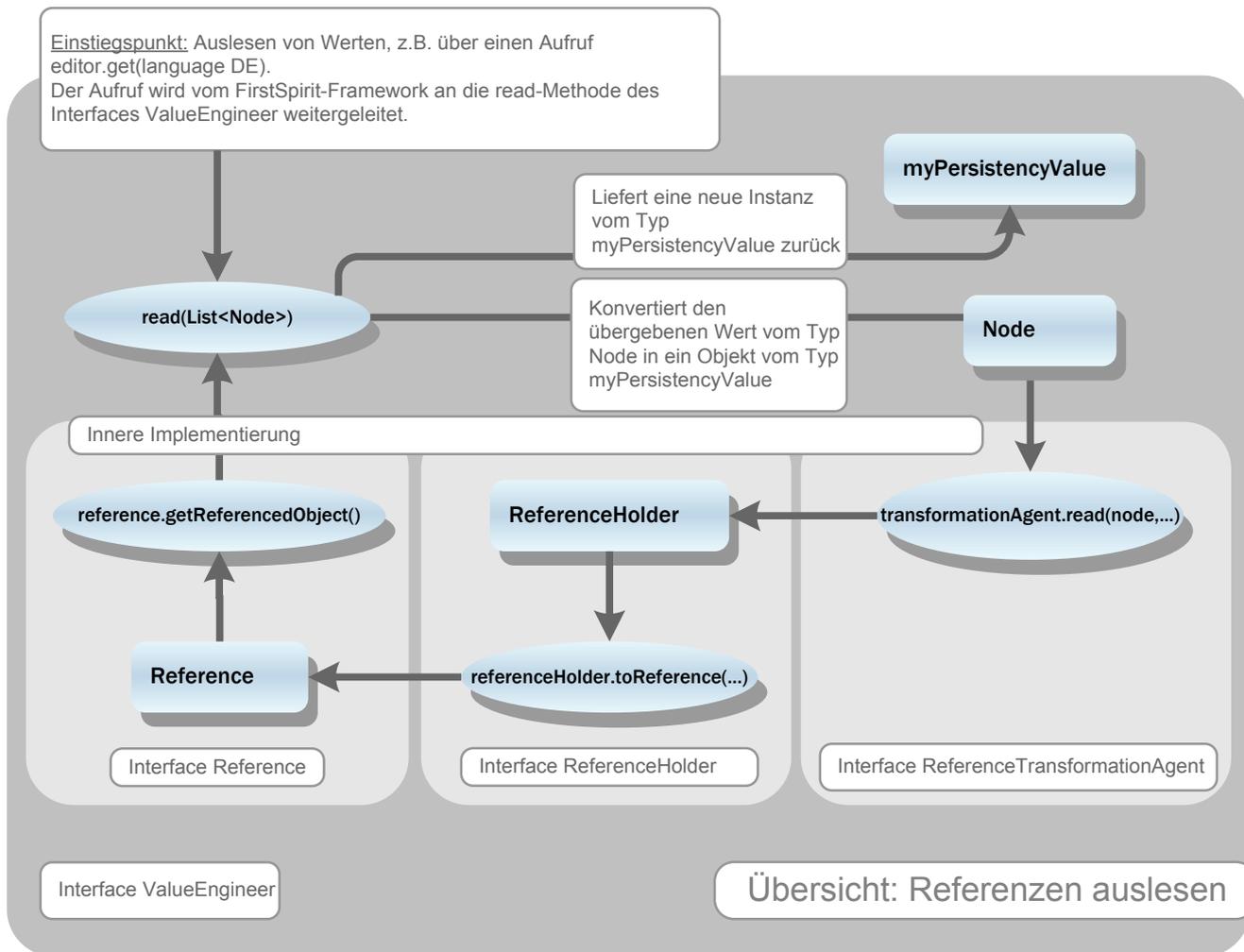


Abbildung 3-3: Lesen von Referenzen



3.12.8.4 Das Interface ReferenceHolder

Package: `de.espirit.firstspirit.client.access.editor`

Eine Instanz vom Typ `ReferenceHolder` ist ein Daten-Container, der benötigt wird, um interne oder externe Referenzen zu erzeugen. Instanzen vom Typ `ReferenceHolder` werden über einen `ReferenceConstructionAgent` erzeugt. Abhängig von den Übergabe-Parametern wird entweder eine Instanz vom Typ `IDProviderReferenceHolder` oder eine Instanz vom Typ `ExternalReferenceHolder` zurückgeliefert (siehe Kapitel 3.12.8.5 Seite 115):

- Eine Instanz vom Typ `IDProviderReferenceHolder` wird zurückgeliefert, wenn der `create`-Methode des Interfaces `ReferenceConstructionAgent` ein Objekt vom Typ `IDProvider` oder `Dataset` übergeben wird.
- Eine Instanz vom Typ `ExternalReferenceHolder` wird zurückgeliefert, wenn der `create`-Methode des Interfaces `ReferenceConstructionAgent` eine externe Ressource übergeben wird.

Das Interface bietet den Zugriff auf folgende Methoden:

- `Reference toReference(@Nullable ChangeCallback callback)`: Eine Instanz vom Typ `ReferenceHolder` hält nur die Informationen über das (interne oder externe) Objekt, das bei ihrer Erzeugung übergeben wurde. Die eigentliche Referenz wird erst durch den Aufruf der Methode `toReference(...)` auf dem `ReferenceHolder` erzeugt und als neue Instanz vom Typ `Reference` zurückgeliefert (zu `Reference` siehe Kapitel 3.12.8.7 Seite 119). Bei diesen Referenzen handelt es sich ausschließlich um ausgehende Kanten, also Referenzen, die von der Eingabekomponente (bzw. von dem dort enthaltenen Wert) auf andere Objekte verweisen.
Die erzeugten Referenzen, werden bei der Implementierung des Aspekts `ReferenceContaining` benötigt (siehe Kapitel 3.12.7.3 Seite 107).
Beim Ändern des referenzierten Objekts in der Eingabekomponente, wird der `ReferenceHolder` vom FirstSpirit-Framework automatisch mit der neuen Referenz aktualisiert. Die Implementierung einer Callback-Funktion ist in diesem Fall nicht notwendig, der betreffende Parameter muss daher nicht übergeben werden:



```
1.    final ExternalReferenceHolder referenceHolder = agent.create(value,
      "custom");
2.    final Reference reference = referenceHolder.toReference(null);
```

Listing 21: Referenzen - Erzeugen einer neuen Referenz (über ReferenceHolder)

Innerhalb des Beispiels (s.o.) wird der `ExternalReferenceHolder` über einen `ReferenceConstructionAgent` (hier: „agent“) erzeugt. Das genaue Vorgehen wird in Kapitel 3.12.8.5 (Seite 115) beschrieben.

Sind bei einer Änderung des referenzierten Objekts innerhalb der Eingabekomponente weitere Implementierungsschritte erforderlich, beispielsweise weil ein externes System (z.B. eine externe Bild-Datenbank) über die Änderung informiert werden muss, kann der Methode `toReference(...)` zusätzlich ein Objekt vom Typ `ReferenceHolder.ChangeCallback` übergeben werden. Über die Methode `void onChange()` wird diese zusätzliche Implementierung bei einer Änderung der Referenzinformation ausgeführt.

```
1.    @NotNull
2.    public List<Reference> collectReferences(@NotNull final
      ExternalReferenceHolder value) {
3.        final String oldUrl = value.getResource();
4.        return Collections.singletonList(value.toReference(new
          ChangeCallback() {
5.            public void onChange() {
6.                changeExternalReference(oldUrl,
              value.getResource());
7.            }
8.        });
9.    });
10. }
11.
12.
13. private void changeExternalReference(final String oldUrl, final String
      newUrl) {
14.     // inform some external system...
15. }
```

Listing 22: Referenzen - ChangeCallback bei der Änderung einer Referenz



3.12.8.5 Das Interface ReferenceConstructionAgent

Package: de.espirit.firstspirit.client.access.editor

Über einen `ReferenceConstructionAgent` können Instanzen vom Typ `ReferenceHolder` erzeugt werden (siehe Kapitel 3.12.8.4 Seite 113). Ein Objekt vom Typ `ReferenceConstructionAgent` kann über den `ValueEngineerContext` geholt werden (siehe Kapitel 3.12.6.4 Seite 101). Dieser Kontext wird Objekten vom Typ `ValueEngineer<T>` während der Erzeugung übergeben. Um innerhalb der `ValueEngineer`-Implementierung Zugriff auf den Kontext zu haben, muss dieser bei der Erzeugung der `ValueEngineer`-Instanz innerhalb der `ValueEngineerFactory`-Implementierung weitergereicht werden (vgl. Beschreibung zu `ValueEngineerFactory` in Kapitel 3.12.6.2 und die Beispiel-Implementierung in Kapitel 3.14.6 Seite 136):

```
1. public class MyValueEngineerFactory implements
    ValueEngineerFactory<T, MyGom> {
2.     ...
3.     public ValueEngineer<T> create(final
        ValueEngineerContext<MyGom> context) {
4.         return new MyValueEngineer(context);
5.     }
6. }
```

Listing 23: Referenzen - ValueEngineerContext übergeben (ValueEngineerFactory)

Der innerhalb des Konstruktors übergebene Kontext kann anschließend in der `ValueEngineer`-Implementierung weiterverwendet werden:

```
1. public class MyValueEngineer implements ValueEngineer<T>,
    ReferenceContaining<T> {
2.
3.     private ValueEngineerContext<MyGom> _context;
4.
5.     public MyValueEngineer(final ValueEngineerContext<MyGom>
        context) {
6.         _context = context;
7.     }
8.     ...
9.
10.    @NotNull
11.    public List<Reference> collectReferences(@NotNull final
        T value) {
12.        final SpecialistsBroker broker =
            _context.getBroker();
13.        ...
14.    }
15.    ...
```

Listing 24: Referenzen – SpecialistsBroker über ValueEngineerContext anfordern



Über den `ValueEngineerContext` kann eine Instanz vom Typ `SpecialistsBroker` angefordert werden. Ein `SpecialistsBroker` bietet über unterschiedliche „Spezialisten“ Zugriff auf bestimmte Dienste oder Informationen. Für das Erzeugen von Referenzen, wird ein Spezialist vom Typ `ReferenceConstructionAgent` benötigt. Dieser kann auf dem `SpecialistsBroker` mithilfe der Methode `<S> S requireSpecialist(SpecialistType<S> type)` angefordert werden:

```
...  
final ReferenceConstructionAgent referenceConstructionAgent =  
    _context.getBroker().requireSpecialist(ReferenceConstructionAgent.  
    TYPE);  
...
```

Listing 25: Referenzen - ReferenceConstructionAgent anfordern

Auf der Instanz vom Typ `ReferenceConstructionAgent` können anschließend Instanzen vom Typ `ReferenceHolder` für die Aufnahme interner oder externer Referenzen erzeugt werden (siehe Kapitel 3.12.8.4 Seite 113).

Das Interface bietet den Zugriff auf folgende Methoden:

- `IDProviderReferenceHolder create(@NotNull IDProvider element, @Nullable String remoteConfigName):`
Die Methode liefert einen Daten-Container vom Typ `IDProviderReferenceHolder` zurück, der das übergebene `FirstSpirit-Element` (vom Typ `IDProvider`) referenziert. Stammt das übergebene Element aus einem Remote-Projekt, muss als weiterer Parameter der symbolische Name des zugehörigen Remote-Projekts übergeben werden. Handelt es sich bei dem übergeben Element um einen `FirstSpirit-Datensatz` (Typ `Dataset`), so muss dieser aus einer „Datenquelle“ stammen, also einer an `FirstSpirit` angebundenen Datenbank (weiterführende Informationen zu Datenquellen siehe `FirstSpirit Handbuch für Entwickler` (Teil 1 - Grundlagen)).
- `ExternalReferenceHolder create(@NotNull String resource, @Nullable String category):`
Analog zu `FirstSpirit-internen` Referenzen, können auch Referenzen auf externe Quellen (z.B.: ein externer URL) angelegt werden. Die Methode liefert einen Daten-Container vom Typ `ExternalReferenceHolder` zurück, der die übergebene externe Ressource referenziert. Optional kann dem `ExternalReferenceHolder` eine Kategorie übergeben werden, z.B. „url“ für Internetadressen oder „email“. Die hier übergebene Kategorie wird im `FirstSpirit-JavaClient` beispielsweise innerhalb des Suchdialogs angezeigt (Menüpunkt:



„Suche nach externen Referenzen“).

```
1.    ...
2.    final SpecialistsBroker broker = _context.getBroker();
3.
4.    final ReferenceConstructionAgent agent =
        broker.requireSpecialist(ReferenceConstructionAgent.TYPE);
5.
6.    final ExternalReferenceHolder referenceHolder = agent.create(value,
        "custom");
7.    ...
```

Listing 26: Referenzen - ReferenceHolder erzeugen (ValueEngineer-Impl.)

3.12.8.6 Das Interface ReferenceTransformationAgent

Package: `de.espirit.firstspirit.client.access.editor`

Das Interface `ReferenceTransformationAgent` stellt Funktionalität bereit, mit deren Hilfe Instanzen vom Typ `ReferenceHolder` in ihre Persistenzform (und umgekehrt) transformiert werden können. Ein `ReferenceTransformationAgent` kann `ReferenceHolder` auf FirstSpirit-Elemente (Typ `IDProvider`), Datensätze (Typ `Dataset`) und externe Ressourcen (z.B. URLs) behandeln (siehe Kapitel 3.12.8.4 Seite 113).

Eine Instanz vom Typ `ReferenceTransformationAgent` kann über einen `SpecialistsBroker` (`SpecialistBroker` siehe Kapitel 3.12.4.1 Seite 95) mithilfe der Methode `requireSpecialist(ReferenceTransformationAgent.TYPE)` angefordert werden:

```
...
final ReferenceTransformationAgent ReferenceTransformationAgent =
    _context.getBroker().requireSpecialist(ReferenceTransformationAgent
    t.TYPE);
...
```

Listing 27: Referenzen - ReferenceTransformationAgent anfordern

Eine neue Instanz vom Typ `SpecialistsBroker` wird über den `ValueEngineerContext` geholt (Beispiel siehe [Listing 23: Referenzen – SpecialistsBroker über ValueEngineerContext anfordern](#)). Dieser Kontext wird Objekten vom Typ `ValueEngineer<T>` während der Erzeugung übergeben. Um innerhalb der `ValueEngineer`-Implementierung Zugriff auf den Kontext zu haben, muss dieser bei



der Erzeugung der ValueEngineer-Instanz innerhalb der ValueEngineerFactory-Implementierung weitergereicht werden ([Listing 23: Referenzen - ValueEngineerContext übergeben \(ValueEngineerFactory\)](#)).

Das Interface bietet den Zugriff auf folgende Methoden:

- `ReferenceHolder read(@NotNull Node node, final Language language, final boolean release)`: Die Methode ist zuständig für das Lesen der Werte aus der Persistenz und die Überführung in ein Objekt vom Typ `ReferenceHolder` (Xml to Object). Dazu wird ein Objekt vom Typ `Node` übergeben, das die Informationen zum referenzierten Element enthält. Außerdem wird der Parameter `language` übergeben, der die Sprache definiert, für die ein Knoten gelesen werden soll, und der Parameter `release`, der definiert ob die Informationen im Freigabestand vorliegen. Ein Objekt vom Typ `Node` ist ein Daten-Container, der einen Namen, Attribute (optional) und entweder einen textuellen Wert oder weitere Objekte vom Typ `Node` (Kind-Knoten) enthält. Die für einen Knoten gespeicherten Werte werden als Instanz vom Typ `ReferenceHolder` zurückgeliefert (siehe Kapitel 3.12.8.4 Seite 113).

```
1. @Nullable
2. public MyPersistencyObject read(@NotNull final List<Node> nodes) {
3.     final SpecialistsBroker broker = _context.getBroker();
4.     final ReferenceTransformationAgent agent =
5.         broker.requireSpecialist(ReferenceTransformationAgent.TYPE);
6.     ExternalReferenceHolder ref = agent.read(nodes.get(0),
7.         _context.getLanguage(), _context.isRelease());
8.     return MyPersistencyObject(ref);
9. }
```

Listing 28: Referenzen - ReferenceTransformationAgent – read(...)

Analog zur Beispiel-Implementierung einer Text-Eingabekomponente in Kapitel 3.14 kann der `ReferenceHolder` auch als Datentyp einer Eingabekomponente über die `read`-Methode der ValueEngineer-Implementierung zurückgeliefert werden (vgl. [Beispiel ValueEngineer-Impl. – Methode read\(...\) – Xml to Object](#)), z.B.:

```
1. @Nullable
2. public ExternalReferenceHolder read(@NotNull final List<Node>
3.     nodes) {
4.     final SpecialistsBroker broker = _context.getBroker();
5.     final ReferenceTransformationAgent agent =
6.         broker.requireSpecialist(ReferenceTransformationAgent.TYPE);
7.     return (ExternalReferenceHolder) agent.read(nodes.get(0),
8.         _context.getLanguage(), _context.isRelease());
9. }
```

Listing 29: Referenzen - ReferenceTransformationAgent – read(...)

- `Node write(@NotNull String tag, @NotNull ReferenceHolder`



holder): Die Methode ist zuständig für die Übernahme der Werte aus der Instanz vom Typ ReferenceHolder in ihre Persistenzform (Object to Xml). Die Methode konvertiert den übergebenen ReferenceHolder dazu in ein Objekt vom Typ Node. Ein Objekt vom Typ Node muss neben seinem textuellen Wert (oder weiteren Objekten vom Typ Node) auch einen Namen besitzen, der hier über den Parameter tag übergeben wird.

Analog zur Beispiel-Implementierung einer Text-Eingabekomponente in Kapitel 3.14 kann der hier zurückgelieferte Node über die write-Methode der ValueEngineer-Implementierung zurückgeliefert werden (vgl. [Beispiel ValueEngineer-Impl. – Methode write\(...\) –Object to Xml](#)), z.B.:

```
1. @NotNull
2. public List<Node> write(@NotNull final ExternalReferenceHolder
   value) {
3.     final SpecialistsBroker broker = _context.getBroker();
4.     final ReferenceTransformationAgent agent =
       broker.requireSpecialist(ReferenceTransformationAgent.TYPE);
5.     final Node node = agent.write("url", value);
6.     return Collections.singletonList(node);
7. }
```

Listing 30: Referenzen - ReferenceTransformationAgent - write(...)

3.12.8.7 Das Interface Reference

Package: de.espirit.firstspirit.access.editor.reference

Eine Abhängigkeit, die der Wert einer Eingabekomponente zu weiteren internen (z.B. einem Medium) oder externen Objekten (z.B. einem URL) besitzt, wird als Referenz bezeichnet. Die ausgehenden Referenzen einer Eingabekomponente (bzw. des dort enthaltenen Wertes) werden vom FirstSpirit Referenzmanager gesammelt und können innerhalb der FirstSpirit Redaktionsumgebung über den Referenzgraph visuell dargestellt werden.

Es wird zwischen folgenden Referenztypen unterschieden:

- **Interne Referenzen:** Dabei handelt es sich um eine Referenz zu einem FirstSpirit-Objekt vom Typ IDProvider (z.B. einer Seitenreferenz oder einem Medium) oder zu einem FirstSpirit-Objekt vom Typ Dataset innerhalb des gleichen FirstSpirit-Projekts.
- **Remote-Referenzen:** Dabei handelt es sich um eine Referenz zu einem verwandten FirstSpirit-Projekt. FirstSpirit unterstützt den Remote-Zugriff auf andere FirstSpirit-Projekte. Über den Remote-Zugriff kann innerhalb des aktuellen Projekts ein Element aus den Verwaltungsbereichen eines weiteren



FirstSpirit-Projekts referenziert und angezeigt werden (z.B. "media:logo, remote:MainProject"). Die Objekte verbleiben dabei physikalisch im Remote-Projekt.

- Externe Referenzen: Dabei handelt es sich um eine Referenz zu einer externen Quelle, beispielsweise einer Website (z.B. "http://www.e-Spirit.com") oder einer Datenbank.

Das Interface bietet den Zugriff auf diese (und weitere) Methoden:

- `boolean isBroken()`: Durch das Löschen von Elementen, die innerhalb des Projekts noch referenziert werden (oder auch durch einen defekten Projektimport), kann es zu ungültigen Referenzen im Projekt kommen. Ein FirstSpirit-Objekt wird über den `UID` (Unique Identifier) und den `UID-Type` (z.B. `MEDIASTORE_LEAF`) referenziert (entsprechende Methoden siehe `FirstSpirit-Access-API`, Interface `IDProvider`). Kann ein referenziertes Objekt über diese Parameter nicht mehr gefunden werden, liefert die Methode `isBroken()` `true` zurück. Die Methode prüft nur interne FirstSpirit-Referenzen und Remote-Referenzen, für externe Referenzen wird immer `false` zurückgeliefert.
- `boolean isRemote()`: Sofern es sich um eine Referenz aus einem verwandten FirstSpirit-Projekt handelt, liefert die Methode `true` zurück.

Weitere Methoden siehe FirstSpirit-Access-API.



3.13 FirstSpirit Security Architektur – Java Web Start (javaws)

Der FirstSpirit JavaClient und die FirstSpirit Server- und Projektkonfiguration werden über eine `jnlp`-Datei, d.h. über Java Web Start ausgeführt. Daraus ergeben sich zunächst einmal Einschränkungen in der Nutzung einiger Funktionalitäten für nicht von e-Spirit signierte Module bzw. in den Jar-Archiven enthaltene Klassen. Java-Programme laufen normalerweise in einer „Sandbox“ ab. D.h. sie haben keinen vollwertigen Zugriff auf den Rechner und dessen Ressourcen, auf dem sie ausgeführt werden, sondern können nur innerhalb der Java-VM (Java Virtual Machine) arbeiten. Der Zugriff auf lokale Ressourcen wie Dateien, Zwischenablage, Netzwerk etc. geschieht über einen Security-Manager. Dieser ist normalerweise entsprechend den Sicherheitsanforderungen konfiguriert. Praktisch heißt das: Programme, die direkt (z.B. von der Konsole) gestartet werden, haben alle Privilegien; Programme, die über eine Netzwerkverbindung gestartet werden (Applets oder Java Web Start-Applikationen), haben erstmal keine Zugriffsprivilegien auf lokale Ressourcen. Die FirstSpirit-eigenen Module sind mit dem „e-Spirit AG“-Schlüssel signiert. Dieser ist wiederum Bestandteil der FirstSpirit-eigenen Security-Policy. Des Weiteren ist der Schlüssel von einer Root-Authority bestätigt, die wiederum dem Java Zertifikat Manager bekannt ist. Im Allgemeinen erhält man bei einer Applikation, die gegen die Sicherheitsrichtlinien verstößt, eine recht informative Meldung (Popup-Dialog). Alle auftretenden Security-Exceptions werden ggf. in der Java Console ausgegeben.



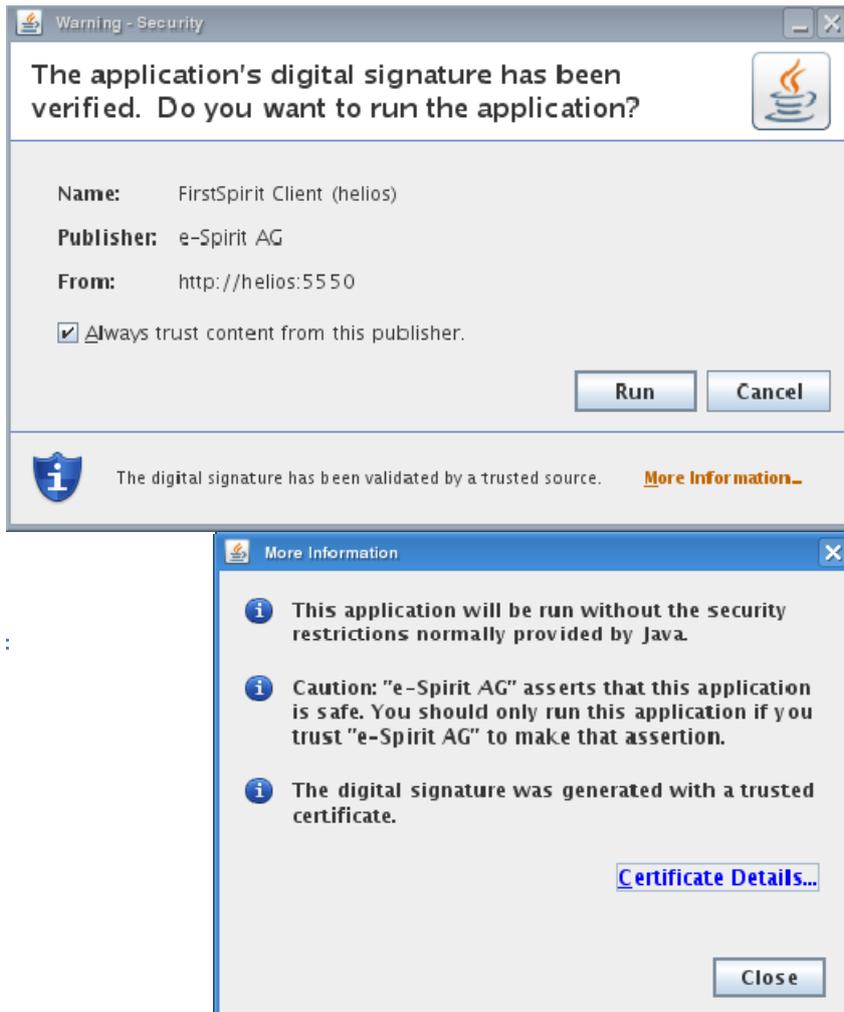


Abbildung 3-4: Java Web Start Security Warning-Dialog

Sollen die in einem Modul enthaltenen Archive und somit deren Klassen mit einem eigenen vertrauenswürdigen Schlüssel signiert werden, um Zugriff auf lokale Ressourcen bzw. sicherheitsrelevante Ressourcen zu erlangen, sind somit zur Ausführung der Web Start-Anwendung zwei vertrauenswürdige Schlüssel innerhalb des FirstSpirit JavaClients vorhanden. Nun sollte man annehmen, dass zwei „recht informative Meldungen“ (Popup-Dialog) beim ersten Starten des JavaClients angezeigt werden, wie es von Applets bekannt ist. Damit sollte die Möglichkeit bestehen, beide Zertifikate zu akzeptieren, d.h. als vertrauenswürdige einzustufen und in den Java Web Start-internen Zertifikat-Cache zu übernehmen. Laut JSR-56⁷ ist dieses nicht der Fall. Es wird immer nur ein Security-Warning-Dialog angezeigt, dieses ist immer das vertrauenswürdige e-Spirit-Zertifikat.

⁷ JSR-56 [9] – Java™ Network Launching Protocol (JNLP) Specification ("Specification")



– Was bedeutet dieses nun für externe Komponenten/Module, die auf sicherheitsrelevante Funktionalitäten zugreifen müssen?

Jar-Archive bzw. die darin enthaltenen Klassen müssen signiert sein, der Signaturschlüssel muss im Java Web Start Zertifikat-Cache auf dem Client importiert werden (importierte Zertifikate überprüfen mittels: `jcontrol` unter dem Reiter Security.)

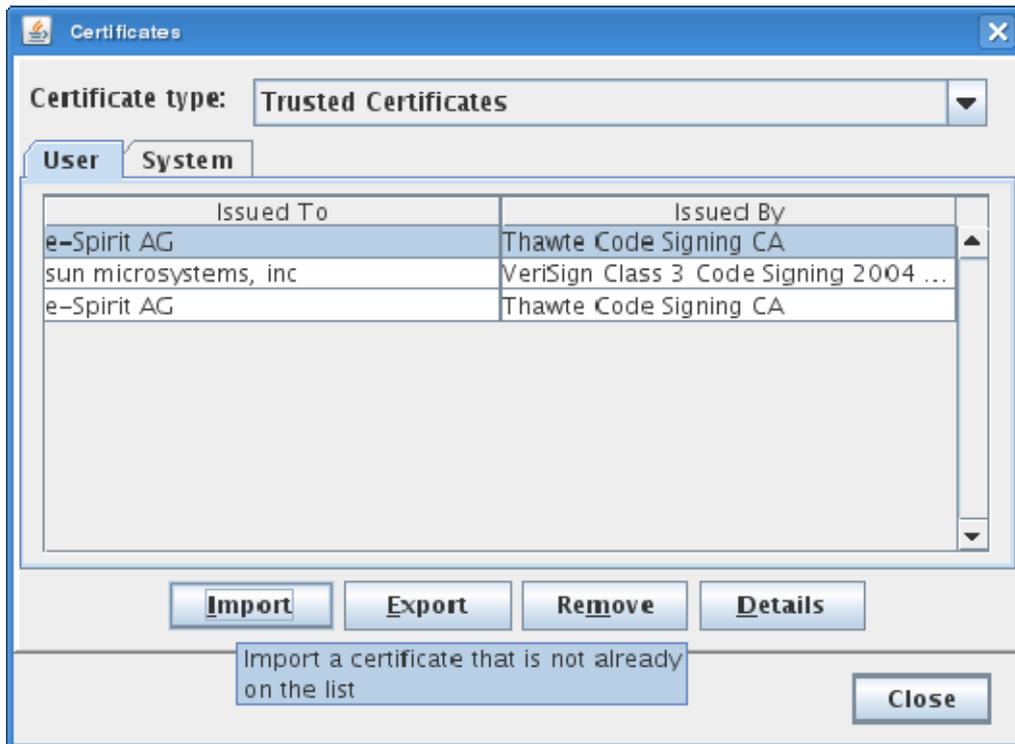


Abbildung 3-5: Java Web Start Zertifikat-Cache / Import von Zertifikaten



Da die zuvor erläuterte Vorgehensweise bei einer Vielzahl von FirstSpirit-Nutzern auf unterschiedlichen Client-Rechnern nicht praktikabel ist, besteht in **FirstSpirit Version 4.1** die komfortable Alternative -- die Verwendung des **FirstSpirit** eigenen **Security-Managers** sowie des **Security-Classloaders**. Die Zuweisung von Rechten auf Modulebene wird über die FirstSpirit Server- und Projektkonfiguration vorgenommen (siehe Kapitel 3.13.1 Seite 124). In der **FirstSpirit Version 4.0** werden alle Module automatisiert durch den FirstSpirit Security-Manager als vertrauenswürdig eingestuft.



3.13.1 Verwendung des FirstSpirit Security-Managers/Classloaders

In der FirstSpirit Server- und Projektkonfiguration kann jedes installierte Modul (mit Ausnahme der FirstSpirit System-Module) optional mit Rechten auf lokale System-Ressourcen ausgestattet werden. Über die Schaltfläche „Konfigurieren“ bei zuvor selektiertem Modul besteht die Option, einem Modul, das sicherheitsrelevante Operationen durchführt, z.B. den Zugriff auf die Zwischenablage (java.awt.AWTPermission ClipboardAccess), zu vertrauen. Diesem Modul können Rechte zur Durchführung der Operationen zugewiesen werden. Dies geschieht intern über den FirstSpirit Security-Manager/Classloader. Der Vorteil dieses Verfahrens ist es, dass nicht jedem Modul voller Zugriff durch den FirstSpirit-Serveradministrator gewährt wird bzw. gewährt werden muss. Es müssen nicht 1..* Zertifikate manuell in den Java Web Start Cache importiert werden. Die Konfigurationsoberfläche zum Setzen der Modulrechte in der FirstSpirit Server- und Projektkonfiguration stellt sich wie folgt dar:

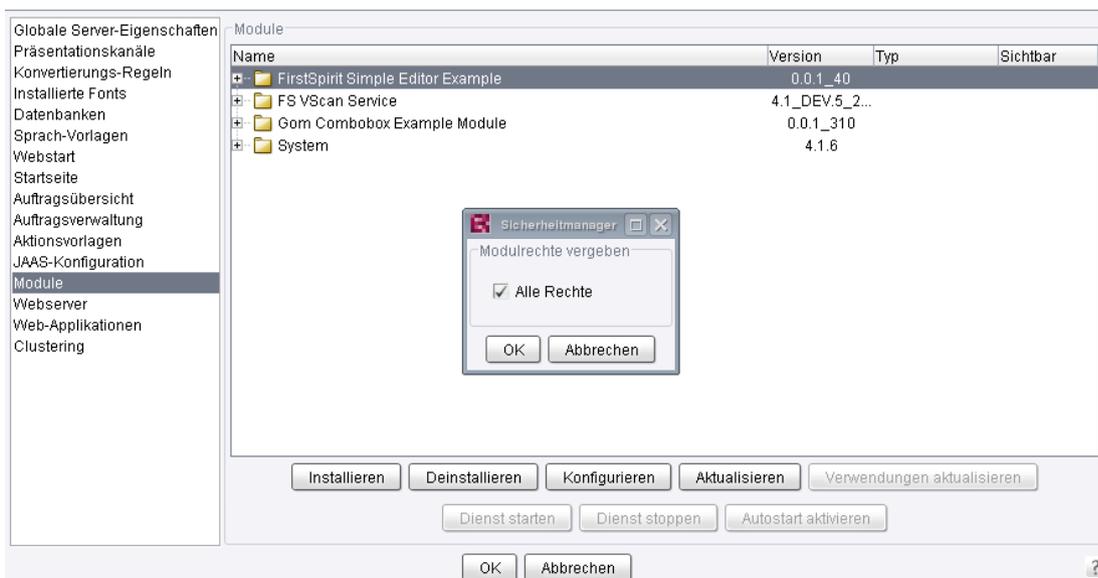


Abbildung 3-6: Setzen der Modulrechte



3.13.2 Zertifikat für Testzwecke erzeugen

Nachfolgend werden beispielhaft die einzelnen Schritte beschrieben, wie sich ein Zertifikat für Testzwecke erzeugen lässt. Die daraus erzeugte Signierung wird von javaws als nicht vertrauenswürdig eingestuft, da es sich nur um ein Zertifikat für eigene Tests handelt.

```
# Create a CA certificate that we will use to sign our certificate
# NOTE: make sure the the organisation name for this cert is
different than the keystore/certificates created later on.
openssl req -x509 -new -out CAcert.crt -keyout CAKey.key -days 365

# Generate a server keystore entry
keytool -genkey -keyalg RSA -alias mycert -dname "CN=localhost,
OU=Test AG, O=Test AG, L=Beauchief, S=Dortmund, C=DE" -keystore
keystore.jks -storepass changeit

# Make a certificate request
keytool -certreq -v -alias mycert -keystore keystore.jks -
storepass changeit -file mycert_request.csr

# Sign the certificate request
openssl x509 -req -in mycert_request.csr -CA CA\CAcert.crt -CAkey
CAKey.key -CAcreateserial -out mycert_response.crt -extfile
/etc/ssl/openssl.cnf -extensions v3_ca -days 365

# View the certificate, Make sure it is version 3
keytool -printcert -file mycert_response.crt

# Add the CA cert to the CA keystore
keytool -import -trustcacerts -alias cacert -file CAcert.crt -
keystore cacerts.jks -storepass changeit

# Add the CA cert to the keystore so that it can find the chain
keytool -import -trustcacerts -alias cacert -file CAcert.crt -
keystore keystore.jks -storepass changeit

# Add the response certificate that has been signed by the CA,
Should get a response of "Certificate reply was installed in
keystore"
keytool -import -alias mycert -file mycert_response.crt -keystore
keystore.jks -storepass changeit
```



3.14 Beispiel: Implementierung einer Eingabekomponente

3.14.1 Übersicht

Zur Veranschaulichung des neuen Komponentenmodells soll die Implementierung einer einfachen Texteingabe-Komponente (basierend auf SwingGadgets) dienen. Das Beispiel erläutert den Zusammenhang zwischen der SwingGadget-Implementierung der Eingabekomponente und der zugehörigen GOM⁸-Form, das Hinzufügen von funktionalen Aspekten zur SwingGadget-Implementierung sowie das Arbeiten mit Werten, die innerhalb der Komponente gespeichert werden können, anhand des neuen Aspekts `ValueHolder`.

- a) Implementierung des Formular-Elements (`GomCustomTextarea`) für die Definition des XML-Identifiers und weiterer Tags und Attribute der neuen Eingabekomponente (`<CUSTOM_TEXTAREA>`), sowie für die Validierung der XML-Repräsentation der Eingabekomponenten im FirstSpirit-JavaClient (siehe Kapitel 3.14.2 Seite 127).
- b) Implementierung einer SwingGadget-Factory (`CustomTextareaSwingGadgetFactory`), die eine neue visuelle Darstellung des SwingGadgets instanziiert und die Implementierung der Eingabekomponente (`CustomTextareaSwingGadget`) mit dem Formular-Element (`GomCustomTextarea`) verknüpft (siehe Kapitel 3.14.3 Seite 129).
- c) Implementierung der SwingGadget-Eingabekomponente (`CustomTextareaSwingGadget`), die einen öffentlichen Konstruktor zur Erzeugung einer neuen visuellen Darstellung des SwingGadgets (ehemals `GuiEditor`) zur Verfügung stellt und alle funktionalen Aspekte der Eingabekomponente implementiert (siehe Kapitel 3.14.4 Seite 130).
- d) Sofern eine Eingabekomponente Werte speichern und bearbeiten kann, müssen Änderungen innerhalb der Komponente nach außen propagiert werden (siehe Kapitel 3.14.5 Seite 133).
- e) Implementierung zur Behandlung der Werte eines SwingGadgets über den Aspekt `ValueHolder` (siehe Kapitel 3.14.7 Seite 137).
- f) Content-Highlighting für eine Eingabekomponenten aktivieren (siehe Kapitel 3.14.8 Seite 142).
- g) Treffermarkierung für die Suchergebnisse einer Eingabekomponente mithilfe des Aspekts `Highlightable` realisieren (siehe Kapitel 3.14.9 Seite 142).

⁸ GUI Object Model



Der vollständige Quellcode ist im Zip-Archiv zum Modul-Entwicklerhandbuch (MODDEV4x_modexamples.zip) im Unterverzeichnis `examples/FS_V4_mod/gadgettextarea` enthalten (vgl. Kapitel 3.10 Seite 53).

3.14.2 GomForm - XML-Repräsentation im JavaClient

Alle GOM⁹-Form-Implementierungen können die abstrakte Basis-Implementierung `AbstractGomFormElement` erweitern (siehe Kapitel 3.11.3.3 Seite 67). Diese stellt grundlegende Funktionalitäten für die meisten Formular-Elemente zur Verfügung, beispielsweise den Parameter `useLanguages` für die Verwendung von Mehrsprachigkeit innerhalb einer Eingabekomponente oder das Attribut `convertEntities` für die Konvertierung von Sonderzeichen in HTML-Code (zur Beschreibung dieser Parameter und Attribute siehe FirstSpirit Online-Dokumentation):

```
1. public class GomCustomTextarea extends AbstractGomFormElement {
2.     ...
3. }
```

Listing 31: Beispiel GOMForm – Erweitern mit der abstrakten Basisimplementierung

Die Implementierung des Formular-Elements definiert den eindeutigen XML-Identifizier für die neue Eingabekomponente, hier `<CUSTOM_TEXTAREA>`, der für die XML-Repräsentation der Eingabekomponente im Vorlagenbereich des FirstSpirit-JavaClients benötigt wird:

```
1. // Editor XML-Identifizier:
   integrates the editor component into the GOM form template
2. public static final String TAG = "CUSTOM_TEXTAREA";
3. ...
4. //--- AbstractGomElement ---//
5.
6. /**
7.  * Return the default tag for a gom element.
8.  *
9.  * @return The elements default tag.
10. */
11. protected String getDefaultTag() {
12.     return TAG;
13. }
```

Listing 32: Beispiel GOMForm – Definition eines XML-Identifiziers

⁹ GOM – GUI Object Model





Bei der Vergabe des Identifiers sollte darauf geachtet werden keine Notation mit „FS_“ zu verwenden, da dies zu Konflikten mit FirstSpirit-Eingabekomponenten führen kann.

Werden für die Konfiguration der Eingabekomponente noch weitere Parameter und Attribute benötigt, so können diese ebenfalls hinzugefügt werden. Soll die neue Eingabekomponente beispielsweise einen Parameter zur Definition der maximal angezeigten Textzeilen erhalten, so kann ein Attribut `rows` und dessen Methoden (Getter/Setter) ergänzt werden:

```
1.  /**
2.   * Number of displayed rows.
3.   */
4.  private PositiveInteger _rows;
5.
6.  /**
7.   * Value of gom attribute 'rows'.
8.   *
9.   * @return Number of displayed rows.
10.  * @see #setRows(de.espirit.firstspirit.common.number.PositiveInteger)
11.  * @see #rows()
12.  */
13.  @GomDoc(description="Number of displayed rows", since="0.0")
14.  @Default("20")
15.  @Nullable
16.  public PositiveInteger getRows() {
17.      return _rows;
18.  }
19.
20.  /**
21.   * Set value of gom attribute 'rows'.
22.   *
23.   * @param value Number of displayed rows.
24.   * @see #getRows()
25.   * @see #rows()
26.   */
27.  public void setRows(@Nullable final PositiveInteger value) {
28.      _rows = value;
29.  }
30.
31.  /**
```



```
32.     * Convenience method to get the number of displayed rows.
33.     * Default value is 20 rows.
34.     * @return Maximum number of rows.
35.     * @see #getRows()
36.     * @see #setRows(de.espirit.firstspirit.common.number.PositiveInteger)
37.     */
38.     public int rows() {
39.         if (_rows != null) {
40.             return _rows.intValue();
41.         }
42.         return 20;
43.     }
```

Listing 33: Beispiel GOMForm – Definition von Attributen

Die Validierung der XML-Tags und -Attribute erfolgt auf dem FirstSpirit-Server durch den Aufruf von `validate()` aus der abstrakten Basis-Implementierung.

Nach der Implementierung des Formular-Elements sieht die XML-Repräsentation der Eingabekomponente folgendermaßen aus:

```
<CMS_MODULE>
  <CUSTOM_TEXTAREA name="myEditor" rows="10">
    <LANGINFOS>
      <LANGINFO lang="*" ..."/>
      <LANGINFO lang="DE" ..."/>
    </LANGINFOS>
  </CUSTOM_TEXTAREA>
</CMS_MODULE>
```

Listing 34: Beispiel - XML-Repräsentation der Eingabekomponente im JavaClient

3.14.3 SwingGadgetFactory - Erzeugen eines typisierten Gadgets

Für jede neue `SwingGadget`-Implementierung muss zunächst eine neue `SwingGadget`-Factory erzeugt werden, die `SwingGadgetFactory<E extends GomElement>` implementiert, wobei `E` das bereits implementierte Formular-Element (`GomCustomTextarea`) ist. Die Factory ruft den öffentlichen Konstruktor der zugehörigen `SwingGadget`-Implementierung auf (siehe Kapitel 3.14.4 Seite 130) und übergibt diesem einen typisierten Kontext. Die `SwingGadget`-Factory (`CustomTextareaSwingGadgetFactory`) verknüpft die Implementierung der Eingabekomponente (`CustomTextareaSwingGadget`) mit dem zugehörigen Formular-Element (`GomCustomTextarea`).



```
1. public class CustomTextareaSwingGadgetFactory implements
   SwingGadgetFactory<GomCustomTextarea> {
2.
3.     @NotNull
4.     public SwingGadget create(@NotNull final
   SwingGadgetContext<GomCustomTextarea> context) {
5.         return new CustomTextareaSwingGadget(context);
6.     }
7. }
```

Listing 35: Beispiel SwingGadgetFactory – Erzeugen eines typ. SwingGadgets

3.14.4 SwingGadgets und die Verwendung von Standard-Aspekten

Alle SwingGadget-Implementierungen müssen SwingGadget implementieren und können, sofern sie das Bearbeiten und Speichern von Werten (EditorValues) ermöglichen, zusätzlich noch Editable und ValueHolder vom Typ T implementieren, wobei T immer der Daten-Container-Typ des zugehörigen EditorValue<T> (hier String) ist.

```
1. public class CustomTextareaSwingGadget implements SwingGadget {
2.     ...
3. }
```

Listing 36: Beispiel SwingGadget – Einbinden des Basistyps SwingGadget

Bzw.:

```
1. public class CustomTextareaSwingGadget implements SwingGadget,
   ValueHolder<String>, Editable {
2.     ...
3. }
```

Listing 37: Beispiel SwingGadget – Einbinden der Aspekte ValueHolder und Editable

Stattdessen kann die SwingGadget-Implementierung auch durch die abstrakte Basis-Implementierung AbstractValueHoldingSwingGadget<T, F extends GomFormElement> erweitert werden, die diese und weitere grundlegende Funktionalität bereits zur Verfügung stellt.

```
1. public class CustomTextareaSwingGadget extends
   AbstractValueHoldingSwingGadget<String, GomCustomTextarea> {
2.     ...
3. }
```

Listing 38: Beispiel SwingGadget – Erweiterung mit der abstr Basisimplementierung



Die SwingGadget-Implementierung stellt einen öffentlichen Konstruktor zur Erzeugung einer neuen visuellen Darstellung der Eingabekomponente (ehemals GuiEditor) zur Verfügung, der von der zugehörigen Factory aufgerufen wird:

```

1.  /**
2.   * Example (MODDEV): Constructs a new CustomTextAreaSwingGadget
3.   * a new displayable text editor representation.
4.   * @param context swing gadget context
5.   */
6.   public CustomTextAreaSwingGadget (final
7.   SwingGadgetContext<GomCustomTextArea> context) {
8.       super (context);
9.   }

```

Listing 39: Beispiel SwingGadget – Konstruktor

Die eigentliche Swing-Komponente, das heißt die visuelle Darstellung der Eingabekomponente wird innerhalb der Methode `JComponent getComponent()` aus dem Interface `SwingComponentProvider` implementiert. Die nachfolgende Implementierung liefert beispielsweise eine mehrzeilige Texteingabe-Komponente (`JTextArea`) mit variablen Scroll-Balken zurück:

```

1.  /* SwingGadget
2.   ===== */
3.  public JComponent getComponent() {
4.      if (_component == null) {
5.          _jTextArea = new JTextArea (getForm().rows(), 20);
6.          _jTextArea.getDocument().addDocumentListener (this);
7.
8.          _component = new JScrollPane (_jTextArea,
9.                                       JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
10.                                      JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
11.      }
12.      return _component;
13.  }

```

Listing 40: Beispiel SwingGadget – Implementierung der Texteingabekomponente

Eine SwingGadget-Implementierung kann ein `ValueHolder<T>` sein. Das bedeutet, die innerhalb der Eingabekomponente gesetzten Werte (`EditorValues`) werden persistent gespeichert (siehe Kapitel 3.12.6 Seite 97). Anders als in FirstSpirit 4 werden die `EditorValues` nicht mehr an die Implementierung der Eingabekomponente übergeben, sondern intern nur noch über den Aspekt `ValueHolder<T>` behandelt (siehe Kapitel 3.12.2.3 Seite 73). Das umliegende FirstSpirit-Gadget-Framework behandelt dann alle weiteren Funktionen,



beispielsweise das Speichern des Wertes.

```
1.  /* ValueHolder
2.  ===== */
3.  public String getValue() {
4.      return _jTextArea.getText();
5.  }
6.
7.  public void setValue(@Nullable final String value) {
8.      final String currentValue = getValue();
9.      if ( ! Objects.equal(currentValue, value)) {
10.         _jTextArea.setText(value);
11.     }
12. }
13.
14. public boolean isEmpty() {
15.     return StringUtil.isEmpty(getValue());
16. }
```

Listing 41: Beispiel SwingGadget - Implementierung des Aspekts ValueHolder

Über den Aspekt `Editable` können die Werte einer Eingabekomponente bearbeitet und geändert werden. Der Aspekt ist ebenfalls ein Bestandteil der abstrakten Basisimplementierung. Die Methode `setEditable(boolean editable)` sperrt oder entsperrt die Komponente zur Bearbeitung der Werte:

```
1.  /* Editable===== */
2.
3.  public void setEditable(final boolean editable) {
4.      _jTextArea.setEditable(editable);
5.  }
```

Listing 42: Beispiel SwingGadget - Implementierung des Aspekts Editable



3.14.5 NotifyValueChange - Änderungen propagieren

Sofern eine Eingabekomponente Werte speichern und bearbeiten kann, müssen Änderungen innerhalb der Komponente nach außen propagiert werden. Andernfalls werden die Änderungen nicht gespeichert. Bei einer Änderung der inneren Komponente muss das äußere FirstSpirit-Gadget-Framework aktiv über die Änderung informiert werden. Dies ist über die Methode `notifyValueChange(final GadgetIdentifizier identifizier)` möglich.

Innerhalb des Beispiels wird der Swing-Komponente zunächst ein `DocumentListener` hinzugefügt, der auf Änderungen des Textdokuments reagiert:

```
1. public JComponent getComponent() {
2.     if (_component == null) {
3.         final int maxRows = getForm().getMaxRows().intValue();
4.         _jTextArea = new JTextArea(maxRows, 20);
5.         _jTextArea.getDocument().addDocumentListener(this);
6.         ...
7.     }
8. }
```

Listing 43: Beispiel Änderungen propagieren - Hinzufügen eines `DocumentListeners`

Die zugehörigen Methoden rufen bei jeder Aktion, die zu einer Änderung des Wertes in der Eingabekomponente führt, die Methode `notifyValueChange(final GadgetIdentifizier identifizier)` auf:

```
1. /* DocumentListener
2.  =====*/
3. public void insertUpdate(final DocumentEvent e) {
4.     notifyValueChange(getGadgetId());
5. }
6.
7. public void removeUpdate(final DocumentEvent e) {
8.     notifyValueChange(getGadgetId());
9. }
10.
11. public void changedUpdate(final DocumentEvent e) {
12.     notifyValueChange(getGadgetId());
13. }
```

Listing 44: Beispiel Änderungen propagieren – Aufruf von `notifyValueChange()`

Welche Aktionen eine Änderung des Wertes hervorrufen, entscheidet dabei jede Komponente selbst. Im Beispiel der Texteingabekomponente führt beispielsweise



das Tippen innerhalb des Texteingabefelds zu einer Änderung des Wertes.

Über den Aufruf von `notifyValueChange(final GadgetIdentifizier identifizier)` teilt die Komponente – hier das `SwingGadget` – dem äußeren Framework mit, dass sich ihr Inhalt geändert hat. Erkennt das Framework, dass ein geänderter Wert gespeichert werden muss, erscheint das Symbol  (Speichern) auf dem Tab des aktuell geöffneten Arbeitsbereichs im FirstSpirit-JavaClient:



Nicht jede Änderung muss auch tatsächlich gespeichert werden. Bleibt der Wert innerhalb der Eingabekomponente unverändert (bezogen auf den zuletzt gespeicherten Wert), beispielsweise weil die ursprüngliche Änderung wieder zurückgesetzt wurde, so muss die Änderung nicht gespeichert werden. Das Symbol  (Speichern) auf dem Tab des Arbeitsbereichs wird wieder ausgeblendet.

Um unnötiges Speichern und die damit verbundene Erzeugung einer neuen Revision eines FirstSpirit-Objekts zu vermeiden, wird bei jeder Änderung im `SwingGadget` zunächst überprüft, ob der geänderte Wert dem zuletzt gespeicherten Wert entspricht (Speichern nicht notwendig) oder eine Änderung vorliegt (Speichern notwendig).

Das FirstSpirit-Gadget-Framework führt dazu bei jeder Änderung (also nach jedem Aufruf von `notifyValueChange(final GadgetIdentifizier identifizier)`) einen „Equals“-Vergleich des zuletzt gespeicherten Wertes mit dem aktuell im `SwingGadget` enthaltenen Wert durch, indem der Wert des `SwingGadgets` über die Methode `getValue()` abgefragt wird. Die Methode `getValue()` liefert ein Objekt vom Typ `T` zurück, wobei `T` immer der Daten-Container-Typ des zugehörigen `EditorValue<T>` (hier `String`) ist.

Abhängig von der Art der Eingabekomponente und des zugehörigen Daten-Container-Typs kann der Aufruf von `getValue()` unvorteilhaft sein. Das gilt, wenn:

- eine Transformation des inneren Modells der Eingabekomponente in den Persistenztyp *teuer ist* oder wenn
- eine Transformation des inneren Modells in den Persistenztyp *nicht möglich ist*.

Für Eingabekomponenten, die komplexe Wertemengen speichern (Bsp. DOM-Editor) oder Eingabekomponenten, die erst zu bestimmten Zeitpunkten einen gültigen Wert enthalten (Bsp. manuelle Eingabe eines Datums), sollten eigene Methoden zur Änderungserkennung implementiert werden. Dazu stehen die



funktionalen Aspekte `ChangeManaging` und `ValueLikening` zur Verfügung:

3.14.5.1 Den Aspekt `ChangeManaging` verwenden

Dieser Aspekt sollte immer dann verwendet werden, wenn ein einfacher „Equals“-Vergleich des zuletzt gespeicherten Wertes mit dem aktuell im `SwingGadget` enthaltenen Wert mithilfe von `getValue()` zu teuer ist und/oder es eine performantere Implementierung zur Änderungserkennung gibt.

Beispiel DOM-Editor: Die FirstSpirit-Eingabekomponente `CMS_INPUT_DOM` implementiert einen Undo-Manager, der alle Änderungen innerhalb der Eingabekomponente seit dem letzten Speichervorgang enthält. Statt dem vom Framework durchgeführten „Equals“-Vergleich, kann hier anhand der vorhandenen Implementierung überprüft werden, ob eine Änderung stattgefunden hat. Ist die Undo-Liste des DOM-Editors leer, muss keine Änderung gespeichert werden.

Um die vom Framework durchgeführte, simple „Equals“-Prüfung und damit den wiederholten Aufruf von `getValue()` zu umgehen, sollte die `SwingGadget`-Implementierung den Aspekt `ChangeManaging` implementieren. Dazu muss der Aspekt `ChangeManaging` innerhalb der `SwingGadget`-Implementierung hinzugefügt und die Methoden des Interfaces implementiert werden.

3.14.5.2 Den Aspekt `ValueLikening` verwenden

Dieser Aspekt sollte immer dann verwendet werden, wenn eine Transformation in das Persistenzformat (beim Aufruf von `getValue()`) nicht zu jedem Zeitpunkt möglich ist. Manche Eingabekomponenten erlauben beispielsweise die manuelle Eingabe eines Wertes, der während der Eingabe keinen gültigen Wert für die Komponente darstellt. Hier darf der Aufruf von `getValue()` erst erfolgen, wenn die Eingabekomponente einen gültigen Wert vom Typ `T` enthält. Das Interface `ValueLikening` ist typisiert, d. h. der zu verwaltende Wertetyp wird über die (Java 5) Generics-Funktionalität innerhalb der Implementierung festgelegt.

Beispiel Datums-Eingabekomponente: Neben der Auswahl eines gültigen Datums aus einem Auswahlfeld (Date Picker) erlaubt die Eingabekomponente auch die manuelle Eingabe eines Datums innerhalb eines Textfelds. Hier kann der einfache „Equals“-Vergleich zur Änderungsfeststellung, der durch das FirstSpirit-Gadget-Framework ausgeführt wird, nicht verwendet werden, da der Wert der Eingabekomponente zu Beginn der Eingabe nicht in ein Objekt vom Typ `Date` umgewandelt werden kann.



Um die vom Framework durchgeführte, einfache „Equals“-Prüfung und damit auch den Aufruf von `getValue()` zu einem ungünstigen Zeitpunkt zu umgehen, muss die `SwingGadget`-Implementierung den Aspekt `ValueLikening<T>` implementieren. Dazu muss der Aspekt `ValueLikening<T>` innerhalb der `SwingGadget`-Implementierung hinzugefügt werden und anschließend die Methode `likenTo(T value)` implementiert werden, wobei `T` immer der Daten-Container-Typ des zugehörigen `EditorValue<T>` (hier `Date`) ist:

```
1.  public class DateSwingGadget extends
    AbstractValueHoldingSwingGadget<Date, GomDate>
    implements ValueLikening<Date> {
2.
3.  public DateSwingGadget(final SwingGadgetContext<GomDate> context) {
4.      super(context);
5.      addAspect(ValueLikening.TYPE, this);
6.  }
7.
8.  /* ValueLikening===== */
9.
10. public boolean likenTo(final Date value) {
11.     if (!_datePanel.isValidFormat()) {
12.         return false;
13.     }
14.     return Objects.equal(_datePanel.getDate(), value);
15. }
```

Listing 45: Beispiel Änderungen propagieren – der Aspekt `ValueLikening`

3.14.6 ValueEngineerFactory

Alle FirstSpirit-Eingabekomponenten speichern ihre Werte in sogenannten `EditorValues`. Eine übersichtliche und stabile Schnittstelle zum Interface `EditorValue<T>` (siehe Kapitel 3.12.6.1 Seite 97), ist das Interface `ValueEngineer<T>` (siehe Kapitel 3.12.6.3 Seite 98). Es stellt Basisfunktionalitäten bereit, die für das Lesen und Schreiben des Persistenztyps der neuen Eingabekomponente (`CustomTextareaSwingGadget`) benötigt werden.

Zur Erzeugung eines neuen Objekts vom Typ `ValueEngineer<T>`, muss zunächst eine Klasse bereitgestellt werden, die das Interface `ValueEngineerFactory<T, F extends GomFormElement>` implementiert, wobei `T` immer der Daten-Container-Typ des zugehörigen `EditorValue<T>` (hier `String`) und `F` das bereits implementierte Formular-Element (`GomCustomTextarea`) ist (siehe Kapitel 3.12.6.2



Seite 97).

Damit das `SwingGadget` mit dem korrekten Persistenztyp des zugehörigen `EditorValues` arbeitet, muss über die Methode `Class<T> getType()` der Persistenztyp (hier: `String`) zurückgeliefert werden.

Die `create`-Methode dient anschließend zur Erzeugung eines neuen, typisierten Objekts vom Typ `ValueEngineer<T>`. Der Methode wird ein typisierter `ValueEngineerContext<F>` übergeben (siehe Kapitel 3.12.6.4 Seite 101), der weitere Informationen über den Status eines Werts enthält, beispielsweise die Zielsprache in der ein Wert gespeichert werden soll.

```
1.  public class TextareaValueEngineerFactory implements
    ValueEngineerFactory<String, GomCustomTextarea> {
2.
3.      public Class<String> getType() {
4.          return String.class;
5.      }
6.
7.      public ValueEngineer<String> create(final
    ValueEngineerContext<GomCustomTextarea> context) {
8.          return new TextareaValueEngineer();
9.      }
10. }
```

Listing 46: Beispiel ValueEngineerFactory – Lesen u. Schreiben des Persistenztyps

3.14.7 ValueEngineer - Werte eines SwingGadgets behandeln

Für die Behandlung der Werte, die innerhalb der neuen Eingabekomponente (`CustomTextareaSwingGadget`) gespeichert und bearbeitet werden können, muss anschließend eine Klasse bereitgestellt werden, die das Interface `ValueEngineer<T>` implementiert, wobei `T` immer der Daten-Container-Typ des zugehörigen `EditorValue<T>` (hier `String`) ist.

```
1.  public class TextareaValueEngineer implements ValueEngineer<String> {
2.      ...
3.  }
```

Listing 47: Beispiel ValueEngineer(Implementierung)

Die Implementierung `TextareaValueEngineer` dient dazu Methoden für das Schreiben und Lesen des Persistenzobjekts der Eingabekomponente bereitzustellen. Dazu müssen die entsprechenden Methoden aus dem Interface



ValueEngineer<T> implementiert werden (siehe Kapitel 3.12.6.3 Seite 98).

Das Laden der Werte (Xml to Object) wird über die Methode `T read(@NotNull List<Node> nodes)` ausgeführt. Dazu wird vom FirstSpirit-Framework automatisch eine Liste von Knoten (Nodes) übergeben. Ein Objekt vom Typ `Node` ist ein Daten-Container, der einen Namen, Attribute (optional) und entweder einen textuellen Wert oder weitere Objekte vom Typ `Node` (Kind-Knoten) enthält. Die Implementierung der Methode für die Beispiel-Komponente ist denkbar einfach, da sie lediglich einen textuellen Wert berücksichtigen muss. Es muss also nur der Wert des ersten (und einzigen) Knotens als `String` zurückgeliefert werden. Über die Methode `void setValue(@Nullable T value)` aus dem Interface `ValueHolder<T>` (vgl. Kapitel 3.12.2.3) wird dieser `String` an das `SwingGadget` geliefert (vgl. [Listing 31: Beispiel SwingGadget - Implementierung des Aspekts ValueHolder](#)).

```
1. /**
2.  * See implementation of {@link #write(String)} - we just
3.  *   * return the text content of the first (and only) node
4.  *   * in the provided list.
5.  */
6. @Nullable
7. public String read(@NotNull final List<Node> nodes) {
8.     return nodes.get(0).getText();
9. }
```

Listing 48: Beispiel ValueEngineer-Impl. – Methode read(...) – Xml to Object

Das Lesen der Werte aus dem `SwingGadget` (Object to Xml) wird über die Methode `T getValue()` aus dem Interface `ValueHolder<T>` (vgl. Kapitel 3.12.2.3) ausgeführt (vgl. [Listing 31: Beispiel SwingGadget - Implementierung des Aspekts ValueHolder](#)). Für die Beispiel-Komponente liefert diese Methode einen `String` zurück, der an die Methode `List<Node> write(@NotNull T value)` weitergereicht wird. Die Methode `write(...)` muss den übergebenen `String` in ein Objekt vom Typ `Node` konvertieren und ihn innerhalb einer Liste von Knoten (Nodes) zurückliefern (Object to Xml). Dazu wird zunächst ein neues Objekt vom Typ `Node`, mit einem spezifizierten Namen (hier: „txt“) und dem übergebenen `String` als textuellem Wert (value) erzeugt. Dieser Knoten wird anschließend innerhalb einer Liste von Knoten zurückgeliefert. Die Methode `write(...)` wird bei jedem Speichervorgang im `JavaClient` aufgerufen. Die Implementierung der `read-` und der `write-Methode` müssen zueinander passen. Das bedeutet, der Wert, der in `write(...)` geschrieben wird, muss in `read(...)` geladen werden können und umgekehrt.

```
1. /**
2.  * We create a list with just one node with name {@code txt} and the
3.  *   * text value as content.
4.  */
```



```
4.     @NotNull
5.     public List<Node> write(@NotNull final String value) {
6.         return Collections.singletonList(Node.create("txt", value));
7.     }
```

Listing 49: Beispiel ValueEngineer-Impl. – Methode write(...) –Object to Xml

Für die Behandlung komplexer Datentypen werden die Methoden `T getEmpty()` und `boolean isEmpty(@NotNull T value)` benötigt. Komplexe Werte sind nie null, möglicherweise jedoch leer (siehe auch „Wertetypen“ in Kapitel 3.12.6.5 (Seite 102). Die Methode `T getEmpty()` erzeugt eine neue, leere Instanz eines Persistenztyps (Leerwert), beispielsweise eine leere Liste. Die Methode `isEmpty(@NotNull T value)` prüft, ob die Instanz eines komplexen Persistenztyps, beispielsweise eine Liste, leer ist oder nicht. Da die Methoden Bestandteil des Interfaces `ValueEngineer<T>` sind, müssen sie auch für einfache Persistenztypen implementiert werden. Analog zur Beispiel-Implementierung kann an dieser Stelle jedoch einfach `null` zurückgeliefert werden.

```
1.     /**
2.      * No special empty value, return just {@link null}.
3.      */
4.     public String getEmpty() {
5.         return null;
6.     }
7.
8.     /**
9.      * No special empty value -> provided value cannot be empty ->
10.     * return {@code false}.
11.     */
12.     public boolean isEmpty(@NotNull final String value) {
13.         return false;
14.     }
```

Listing 50: Beispiel ValueEngineer-Impl. – Leerwert-Behandlung

Desweiteren muss die Methode `T copy(@NotNull T original)` implementiert werden, die eine neue Instanz des übergebenen Persistenztyps mit identischen Inhalten (Kopie) erzeugen soll. Da es sich beim Datentyp `String` der Beispiel-Implementierung, um einen einfachen, unveränderlichen Datentypen handelt, kann an dieser Stelle auch die Original-Instanz zurückgeliefert werden. Für komplexe Wertetypen sollte aber immer eine Kopie der Original-Instanz zurückgeliefert und darauf geachtet werden, dass eine Änderung der Original-Instanz keine Auswirkung auf die Kopie hat:



```
1.  /**
2.   * {@link String} instances are immutable, so we can return the
3.   * original value.
4.   */
5.   @NotNull
6.   public String copy(@NotNull final String original) {
7.       return original;
8.   }
```

Listing 51: Beispiel ValueEngineer-Impl. – Kopieren des Persistenztyps

Abhängig von der Implementierung des `SwingGadgets`, können Werte sprachabhängig gespeichert werden. In diesem Fall muss bei der weiteren Implementierung auch die gewünschte Zielsprache berücksichtigt werden. Die Sprachinformationen (und weitere Informationen zum Status eines Wertes) können über den `ValueEngineerContext` ermittelt werden (siehe Kapitel 3.12.6.4 Seite 101). Für die Ermittlung der Zielsprache ist das beispielsweise der Aufruf `ValueEngineerContext.getLanguage()`.

Um die Stabilität des Interfaces `ValueEngineer<T>` zu gewährleisten, wird alle Funktionalität, die über die bisher implementierte Basisfunktionalität hinausgeht, über Aspekte realisiert (siehe Kapitel 3.12.7 Seite 102). Die unterschiedlichen Aspekt-Typen stehen als Interface zur Verfügung. Die `ValueEngineer`-Implementierung muss die gewünschten Aspekte lediglich in der Methode `<T> T getAspect(@NotNull AspectType<T> aspect)` bekanntgeben und falls erforderlich, die entsprechenden Methoden des Interfaces implementieren. Das umliegende FirstSpirit-Gadget-Framework behandelt dann automatisch alle weiteren Funktionen.

Die Beispiel-Implementierung soll eine Suchfunktion mit Treffermarkierung unterstützen (siehe Kapitel 3.14.9 Seite 142). Dazu müssen die im `ValueEngineer` enthaltenen Werte indiziert und Treffer, die dem definierten Suchmuster entsprechen, als Liste zurückgeliefert werden. Diese Funktionalität wird über den Aspekt `MatchSupporting` bereitgestellt (siehe Kapitel 3.12.7.1 Seite 104).

Durch Implementierung der Methode `getAspect(...)` muss zunächst eine aspektorientierte Instanz des Persistenztyps zurückgeliefert werden:

```
1.  /**
2.   * Supported aspects:
3.   * <ul>
4.   * <li>{@link MatchSupporting}</li>
5.   * </ul>
```



```
6.     */
7.     public <T> T getAspect(@NotNull final AspectType<T> aspect) {
8.         if (aspect == MatchSupporting.TYPE) {
9.             return aspect.cast(this);
10.        }
11.        return null;
12.    }
```

Listing 52: Beispiel ValueEngineer-Impl. – Erzeugen einer aspektorientierten Instanz

Zusätzlich müssen die Methoden `getStringForIndex(T value)` und `getMatches(Request request, T value)` aus dem Interface `MatchSupporting` implementiert werden. Die erste Methode muss den übergebenen Wert in einen `String` transformieren, damit die Inhalte für die Suche indiziert werden können. Da die Beispiel-Implementierung bereits mit diesem Datentyp arbeitet, ist es hier ausreichend, den übergebenen Wert zurückzuliefern. Die zweite Methode soll eine Liste von Treffern liefern, die dem übergebenen Suchmuster entsprechen. Die Liste wird zur Visualisierung der Suchtreffer innerhalb des `SwingGadget` (siehe Kapitel 3.14.9 Seite 142) an die Methode `highlight` aus dem Interface `Highlightable` weitergereicht (Beispiel siehe [Listing 47: BeispielSwingGadget – der Aspekt Highlightable](#)).

```
1.     // MatchSupporting<String> aspect
2.
3.     @NotNull
4.     public String getStringForIndex(@NotNull final String value) {
5.         return value;
6.     }
7.
8.     @NotNull
9.     public List<? extends Match> getMatches(@NotNull final Request
    request, @NotNull final String value) {
10.        final List<Match> matches = new ArrayList<Match>();
11.        for (final PatternClause clause : request.getClauses()) {
12.            final Pattern pattern = request.toPattern(clause);
13.            final Matcher matcher = pattern.matcher(value);
14.            final boolean match = matcher.find();
15.            if (match) {
16.                matches.add(new Match(clause,
    matcher.start(), matcher.end()));
17.                while (matcher.find()) {
18.                    matches.add(new Match(clause,
    matcher.start(), matcher.end()));
```



```
19.         }
20.         } else if (request.getJunction() == Junction.AND) {
21.             return Collections.emptyList();
22.         }
23.     }
24.     return matches;
25. }
```

Listing 53: Beispiel ValueEngineer-Impl. – Value-Aspekt MatchSupporting

3.14.8 Content-Highlighting für eine Komponente aktivieren

Um eine neue FirstSpirit-Eingabekomponente mit der Funktionalität „Content Highlighting“ auszustatten, ist keine Implementierung durch den Komponenten-Entwickler erforderlich. Jede FirstSpirit-Eingabekomponente, die einen Swing-Fokus bereitstellt, kann automatisch in das „Content Highlighting“ integriert werden. Innerhalb des Projekts ist dazu lediglich eine Anpassung der Vorlagen erforderlich:

Beispiel (Ausgabekanal html für die Eingabekomponenten CUSTOM_TEXTAREA):

```
$CMS_RENDER(template:"WEBeditInplaceAttributeBegin",
name:"custom_textarea")$
<br>$CMS_VALUE(custom_textarea, default:"-WERT NICHT GESETZT!-")$
$CMS_RENDER(template:"WEBeditInplaceAttributeEnd")$
```

Listing 54: Beispiel – ContentHighlighting aktivieren

Weitere Informationen zum Content Highlighting siehe FirstSpirit Online-Dokumentation.

3.14.9 Aspekt Highlightable - Treffermarkierung für die Suche

Neben den Standardaspekten, können einer SwingGadget-Implementierung noch eine Vielzahl funktionaler Aspekte hinzugefügt werden (siehe Kapitel 3.12.2 Seite 69). Anhand des Beispiels der Texteingabekomponente wird an dieser Stelle die Verwendung des Aspekts `Highlightable` gezeigt.

Zielsetzung: Die Eingabekomponente soll eine „Treffermarkierung“ für Suchergebnisse erhalten. Dazu sollen nach einer Suche, die Fundstellen folgendermaßen markiert werden:

- Die Komponente die den Treffer enthält, wird innerhalb des Formulars hervorgehoben
- Der Treffer wird innerhalb der Komponente markiert
- Liegt der Treffer außerhalb des Sichtbereichs ändert sich der Sichtbereich, um



den Treffer anzuzeigen.

Die Markierung auf Komponentenebene erfolgt automatisch durch das FirstSpirit-Gadget-Framework und erfordert kein weiteres Zutun des Komponententwicklers. Weitere Funktionen, wie das Hervorheben des Treffers innerhalb der Komponente und ggf. das Ändern des Sichtbarkeitsbereichs (Scrolling), müssen über den Aspekt `Highlightable` hinzugefügt werden. Diese Funktionen können vom FirstSpirit-Gadget-Framework nicht allgemein realisiert werden, da dazu eine genaue Kenntnis des inneren Aufbaus der Komponente erforderlich ist. Der Komponententwickler muss aber nur genau diese Funktionen (beispielsweise Scrolling) realisieren und bekommt alle dazu notwendigen Informationen über die Aspekt-Schnittstelle zur Verfügung gestellt.

Um den Aspekt zu verwenden, muss die `SwingGadget`-Implementierung den Aspekt `Highlightable` implementieren. Dazu muss der Aspekt `Highlightable` innerhalb des öffentlichen Konstruktors der `SwingGadget`-Implementierung hinzugefügt und die Methode `highlight(List<? extends Match> matches)` implementiert werden. Die Methode `highlight(...)` implementiert die Infrastruktur, um den Treffer auch innerhalb der Komponente hervorzuheben und ggf. durch Scrolling sichtbar zu machen. Die Liste der Treffer (`matches`), die der Methode übergeben wird, wird von der `ValueEngineer`-Implementierung der Eingabekomponente geliefert (siehe Kapitel 3.14.7 Seite 137). Diese muss den Value-Aspekt `MatchSupporting` implementieren (Beispiel siehe [Listing 44: Beispiel ValueEngineer-Impl. – Value-Aspekt MatchSupporting](#)).

```
1. public class CustomTextareaSwingGadget extends
   AbstractValueHoldingSwingGadget<String, GomCustomTextarea>
   implements DocumentListener, Highlightable {
2.
3.     /**
4.      * Example (MODDEV): Constructs a new CustomTextareaSwingGadget - a new
       displayable text editor representation.
5.      *
6.      * @param context swing gadget context
7.      */
8.     public CustomTextareaSwingGadget(final
       SwingGadgetContext<GomCustomTextarea> context) {
9.         super(context);
10.        addAspect(Highlightable.TYPE, this);
11.    }
12.
13.    /* Highlightable
```



```
14.  =====*/
15.  public void highlight(final List<? extends Match> matches) {
16.      final Highlighter highlighter = _jTextArea.getHighlighter();
17.      final Highlighter.Highlight[] highlights =
18.          highlighter.getHighlights();
19.
20.      // remove all highlights
21.      highlighter.removeAllHighlights();
22.
23.      // restore foreign highlights
24.      for (final Highlighter.Highlight highlight : highlights) {
25.          try {
26.              if (!(highlight.getPainter() instanceof
27.                  DefaultHighlighter.DefaultHighlightPainter) ||
28.                  ((DefaultHighlighter.DefaultHighlightPainter)
29.                   highlight.getPainter()).getColor() !=
30.                   Color.YELLOW) {
31.                  highlighter.addHighlight(highlight.getStartOffset(),
32.                                          highlight.getEndOffset(), highlight.getPainter());
33.              }
34.          } catch (BadLocationException e) {
35.              // ignore
36.          }
37.      }
38.
39.      // add new highlights
40.      if (!matches.isEmpty()) {
41.          final Highlighter.HighlightPainter painter = new
42.              DefaultHighlighter.DefaultHighlightPainter(Color.YELLOW);
43.          try {
44.              Rectangle rect = null;
45.              for (final Match match : matches) {
46.                  final int a = match.getBegin();
47.                  final int b = match.getEnd();
48.                  highlighter.addHighlight(a, b, painter);
49.                  final Rectangle r1 = _jTextArea.modelToView(a);
50.                  if (rect == null) {
51.                      rect = r1;
52.                  } else {
53.                      rect.add(r1);
54.                  }
55.              }
56.
57.              if (rect != null) {
```



```

50.         final Rectangle r2 = _jTextArea.modelToView(b);
51.         if (r2 != null) {
52.             rect.add(r2);
53.         }
54.     }
55. }
56.     if (rect != null) {
57.         _jTextArea.scrollRectToVisible(rect);
58.     }
59.     } catch (final BadLocationException e) {
60.         // ignore
61.     }
62. }
63. }
64. }

```

Listing 55: BeispielSwingGadget – der Aspekt Highlightable

3.14.10 Modul- und Komponenten-Deskriptor

```

1.     <!DOCTYPE module SYSTEM "http://www.FirstSpirit.de/module.dtd">
2.     <module>
3.         <name>FirstSpirit Swing Gadget Example (Textarea)</name>
4.         <version>@VERSION@</version>
5.         <description>FirstSpirit Swing Gadget Module Example</description>
6.         <vendor>e-Spirit AG</vendor>
7.         <components>
8.             <public>
9.                 <name>CUSTOM_TEXTAREA</name>
10.                <description>SwingGadget Editor component.</description>
11.            <class>de.espirit.firstspirit.module.GadgetSpecification </class>
12.            <configuration>
13.                <gom>de.espirit.firstspirit.opt.examples.gadgettextarea.GomCustom
14.                Textarea</gom>
15.                <factory>de.espirit.firstspirit.opt.examples.gadgettextarea.Custom
16.                TextareaSwingGadgetFactory</factory>
17.                <value>de.espirit.firstspirit.opt.examples.gadgettextarea.Custom
18.                TextareaEditorValueImpl</value>
19.            </configuration>
20.
21.            <resources>
22.                <resource>lib/custom-textarea-example-0.0.1_00.jar
23.            </resource>

```



```
20.         </resources>  
21.         </public>  
22.     </components>  
23. </module>
```

Listing 56: Beispiel - Modul- und Komponenten-Deskriptor



3.14.11 Ant build.xml – Komplettes Beispiel

```
1.    <?xml version="1.0" encoding="ISO-8859-1"?>
2.    <!--
3.         to generate fsm module file call
4.         ant -f
5.         src/de/espirt/firstspirit/opt/example/editor/simple/build.xml [TARGET]
6.
7.         Targets:
8.             jar            builds the jar file.
9.             fsm            builds the module file.
10.            deploy        deploys the module file.
11.
12.    -->
13.    <project name="SimpleEditorExample" default="fsm">
14.
15.        <property environment="env"/>
16.        <property name="env.PROJECT_HOME"
17.            value="/path/to/your/module/project_home"/>
18.        <property name="path"
19.            value="de/espirt/firstspirit/opt/example/editor/simple"/>
20.        <property name="classes-dir"
21.            value="${env.PROJECT_HOME}/classes.modules"/>
22.        <property name="main-jar-file" value="fs-seditor-example.jar"/>
23.        <property name="main-fsm-file" value="fs-seditor-example.fsm"/>
24.        <property name="main-tmp-dir" value="${env.PROJECT_HOME}/tmp/fs-
25.            seditor-example" />
26.
27.        <path id="class.path">
28.            <pathelement path="${env.PROJECT_HOME}/classes"/>
29.        </path>
30.
31.        <fileset id="classes.main" dir="${classes-dir}">
32.            <include
33.                name="de/espirt/firstspirit/opt/example/editor/simple/*.class"/>
34.        </fileset>
35.
36.        <target name="main-jar" description="Builds all jar files.">
37.            <mkdir dir="${classes-dir}"/>
38.            <javac debug="on" srcdir="${env.PROJECT_HOME}/src"
39.                destdir="${classes-dir}" includes="${path}/*" target="1.5" source="1.5">
40.            <classpath refid="class.path"/>
41.        </target>
42.    </project>
```



```
34.         </javac>
35.         <jar jarfile="${env.PROJECT_HOME}/${main-jar-file}"
    excludes="**/*.dependency" duplicate="preserve" keepcompression="true">
36.             <fileset refid="classes.main"/>
37.         </jar>
38.     </target>
39.     <target name="deploy" depends="fsm" description="Deploys the
    module file (*.fsm) to the server.">
40.         <copy file="${env.PROJECT_HOME}/${main-fsm-file}"
    todir="${env.CMS_SERVER_HOME}/data/modules" overwrite="true"/>
41.     </target>
42.     <target name="fsm" depends="main-fsm" description="Builds all
    module files (*.fsm)."/>
43.     <target name="main-fsm" depends="main-jar " description="Builds
    the main module file (*.fsm).">
44.         <mkdir dir="${main-tmp-dir}/lib"/>
45.         <mkdir dir="${main-tmp-dir}/META-INF"/>
46.         <copy file="${env.PROJECT_HOME}/${main-jar-file}"
    todir="${main-tmp-dir}/lib" preservelastmodified="true" overwrite="true" />
47.         <copy file="module.xml" tofile="${main-tmp-dir}/META-
    INF/module.xml" overwrite="true">
48.             <filterset>
49.                 <filter token="VERSION" value="1.1.1"/>
50.             </filterset>
51.         </copy>
52.         <jar jarfile="${env.PROJECT_HOME}/${main-fsm-file}"
    basedir="${main-tmp-dir}" excludes="**/*.dependency" duplicate="preserve"
    keepcompression="true"/>
53.     </target>
54. </project>
```

Listing 57: ant build.xml mit den Targets main-jar, main-fsm, deploy



3.15 Komponentenlose Modul-Implementierung (JDBC-Connector-Modul)

Um beispielsweise einen MySQL-JDBC-Treiber zur Nutzung innerhalb von FirstSpirit einzubinden oder zu aktualisieren, ist es erforderlich, den FirstSpirit-Server herunterzufahren bzw. einen Neustart durchzuführen. Besteht außerdem die Anforderung, z.B. zwei verschiedene MySQL-JDBC-Treiber für unterschiedliche Datenbankserver mit abweichenden Treiber-Versionen parallel zu betreiben, wird immer der erste von der JVM gefundene Treiber geladen (System-Classloader, siehe auch Kapitel 2.10 Seite 28).

Für den ersten der beiden zuvor beschriebenen Anwendungsfälle bietet sich die Implementierung einer Bibliothek-Modul-Komponente an (siehe Kapitel 3.16 Seite 153) oder, wie in diesem Kapitel beschrieben, als komponentenlose Modul-Umsetzung, wobei der MySQL-Connector hier als Modul-Ressource außerhalb des `<components>`-Elements im Modul-Deskriptor definiert wird und somit über den FirstSpirit Modul-Classloader erreichbar ist und nicht im VM-System-Classloader landet (siehe auch Kapitel 2.10 ‚Classloading‘ Seite 28). Dadurch ergibt sich die Möglichkeit, mehrere Module (bspw. Module MySQL3 und Module MySQL5) mit unterschiedlichen Treiberversionen zu erstellen und diese anschließend parallel zu betreiben, wie im zweiten Anwendungsfall beschrieben. Grundlegend gilt: die Modul-Ressourcen sind systemweit verfügbar. D.h., sie werden in ein temporäres Verzeichnis (eines pro Modul) entpackt und sind dann über den Server-ClassLoader erreichbar. Das führt dazu, dass bei Namensgleichheit von Dateien das erste Modul/Datei/Klasse gewinnt. Das ist in der VM exakt genau so: sind mehrere Jars im ClassPath vorhanden, gewinnt immer die erste gefundene Ressource.

Vorteile der Integration eines Datenbank-Treibers als Modul-Implementierung:

1. Der MySQL-JDBC Treiber kann im laufenden Betrieb des FirstSpirit-Servers nachgeladen werden, der Server muss somit nicht heruntergefahren/neugestartet werden.
2. Es können Treiber in unterschiedlichen Versionen vorliegen und parallel zur Verwendung kommen.





Die Beispiele, die eine JDBC-Modul-Library verwenden, werden aus lizentechnischen Gründen ohne die entsprechende JDBC-Bibliothek ausgeliefert. Diese muss vom jeweiligen Distributor heruntergeladen und im lib-Verzeichnis des Moduls abgelegt werden. Dazu muss die txt-Datei, die als Platzhalter dient, durch die entsprechende Bibliothek ersetzt werden, z.B. „mysql-connector-java-3.1.14-bin.jar.txt“ im Verzeichnis Library/MySQL03/lib durch „mysql-connector-java-3.1.14-bin.jar.“

Um aus einem JDBC-Treiber eine FirstSpirit-Modul zu erstellen genügen wenige Schritte (hier am Beispiel von „mysql-connector-java-3.1.14-bin.jar“):

1. Erstellen des Modul-Deskriptors – module.xml

```

1.      <!DOCTYPE module SYSTEM "http://www.FirstSpirit.de/module.dtd">
2.      <module>
3.          <name>MySQL</name>
4.          <version>3.1.14</version>
5.          <description>JDBC Driver for MySQL (MySQL Connector/J
3.1.14)</description>
6.          <vendor>MySQL-AB</vendor>
7.          <resources>
8.              <resource>lib/mysql-connector-java-3.1.14-bin.jar</resource>
9.          </resources>
10.         <components/>
11.     </module>

```

Listing 58: Modul-Deskriptor ohne Komponenten – module.xml

2. Anlegen der Modul-Verzeichnisstruktur

MySQL	4,0 KB	Folder
lib	4,0 KB	Folder
mysql-connector-java-3.1.14-bin.jar	448,3 KB	Java Archive
META-INF	4,0 KB	Folder
module.xml	277 B	XML Document

Abbildung 3-7: Anlegen der Verzeichnisstruktur für das JDBC-Library-Modul



3. Erstellen der Modul-Archiv-Datei

Entweder über das Kommando:

```
$ zip -r MySQL.fsm .
adding: META-INF/ (stored 0%)
adding: META-INF/module.xml (deflated 40%)
adding: lib/ (stored 0%)
adding: lib/mysql-connector-java-3.1.14-bin.jar (deflated 4%)
```

oder über die grafische Benutzeroberfläche des bevorzugten Zip-Werkzeugs. Sollte es nicht möglich sein, die aus dem Zip-Vorgang resultierende Datei direkt als *.fsm - MySQL.fsm zu deklarieren, kann die MySQL.zip-Datei einfach umbenannt werden in MySQL.fsm.

4. Das neu erstellte Modul über die FirstSpirit-Anwendung zur Server- und Projektkonfiguration installieren

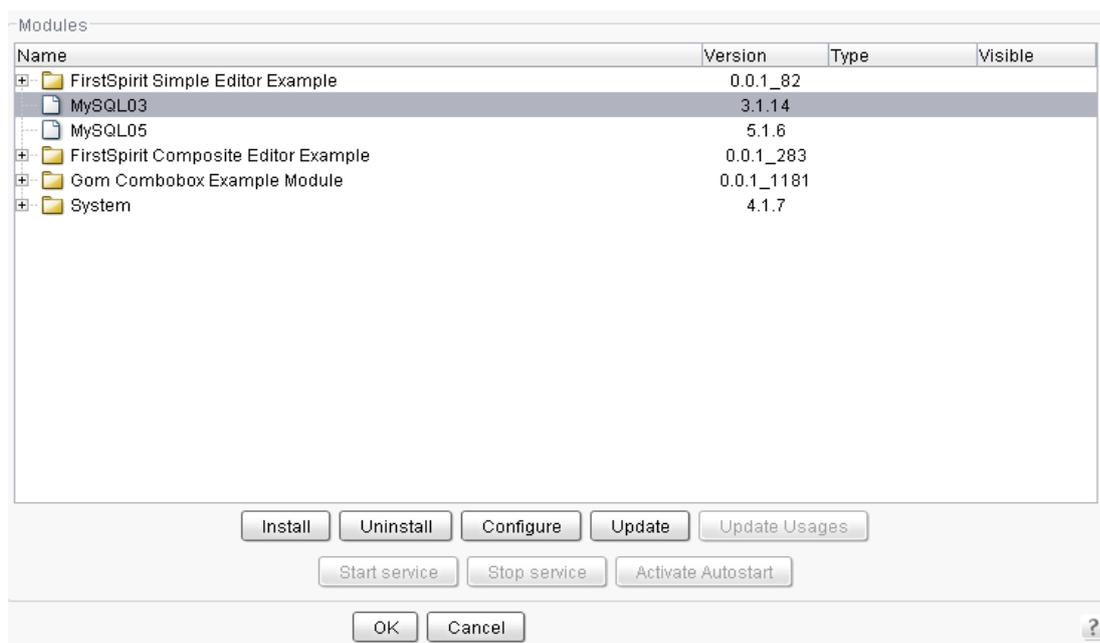


Abbildung 3-8: Komponentenloses Modul – Installation

5. Verwendung des Modul-Treibers

In den Layer-Eigenschaften der entsprechenden Datenbank den Eintrag `module=MySQL03` (Name siehe `module.xml`, Element `<name>MySQL</name>`) ergänzen:



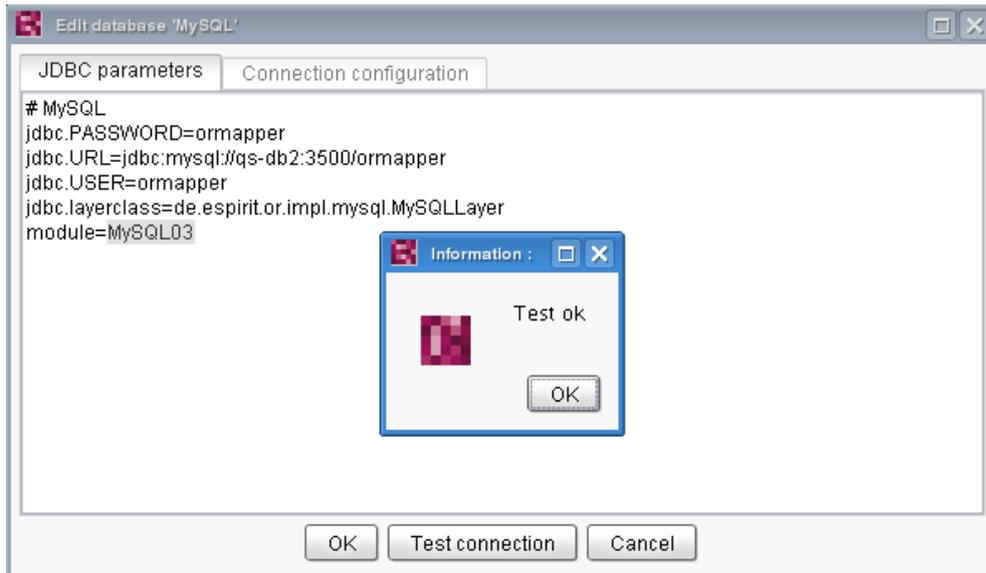


Abbildung 3-9: Verwendung eines Modul-JDBC-Treibers (MySQL) – mysql-connector-java-3.1.14-bin.jar

Um die mysql-connector Version 5 zu nutzen, ist in den Layer-Eigenschaften der entsprechende Datenbank der Eintrag `module=MySQL03` auf `module=MySQL05` zu ändern:

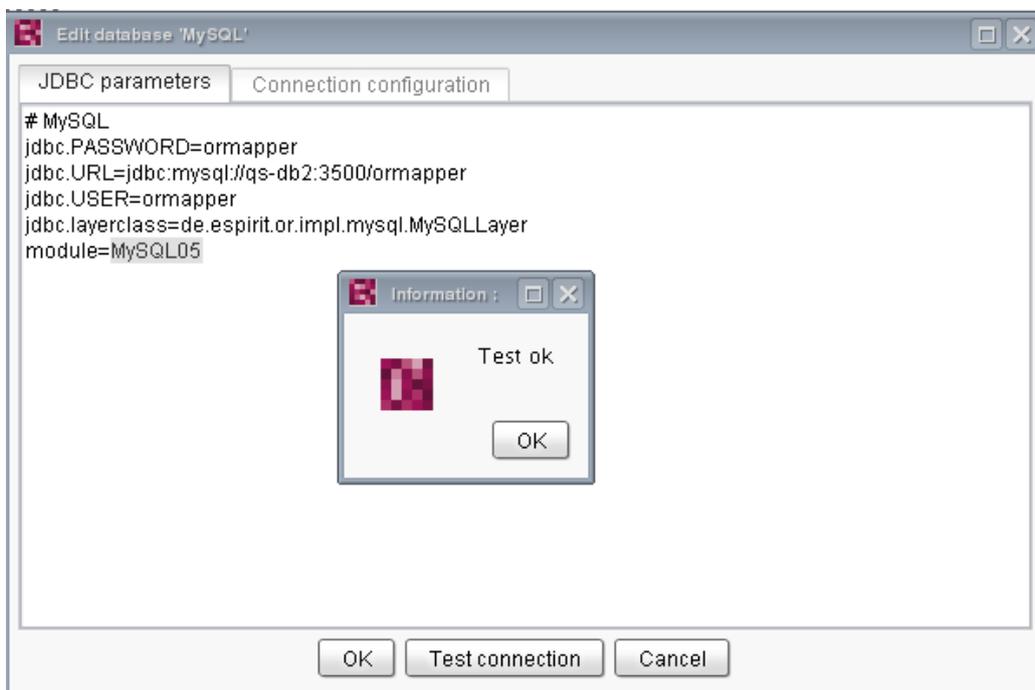


Abbildung 3-10: Verwendung eines Modul-JDBC-Treibers (MySQL) – mysql-connector-java-3.1.14-bin.jar



3.16 Implementierung einer Bibliothek- (Library) Komponente (JDBC-Treiber-Modul)

Hier die gleiche Implementierung wie zuvor unter ‚Komponentenlose Modul-Implementierung (JDBC-Connector-Modul)‘ beschrieben, nur als FirstSpirit-Modul Library-Komponente:

1. Erstellen des Modul-Deskriptors – module.xml

```
1. <!DOCTYPE module SYSTEM "http://www.FirstSpirit.de/module.dtd">
2. <module>
3.     <name>MySQL</name>
4.     <version>0.0.1</version>
5.     <description>JDBC Driver for MySQL</description>
6.     <vendor>MySQL-AB</vendor>
7.     <components>
8.     <library>
9.         <name>JDBC Driver for MySQL - Connector/J</name>
10.        <version>3.1.14</version>
11.        <description>JDBC Driver for MySQL (MySQL Connector/J
12.        3.1.14)</description>
13.        <resources>
14.            <resource>lib/mysql-connector-java-3.1.14-
15.            bin.jar</resource>
16.        </resources>
17.    </library>
18. </components>
19. </module>
```

Listing 59: Modul-Deskriptor mit Library-Komponente – module.xml

2., 3., 5. Die weiteren Schritte sind identisch zu Kapitel 3.15 Seite 149.

4. Das neu erstellte Modul über die FirstSpirit-Anwendung zur Server- und Projektkonfiguration installieren



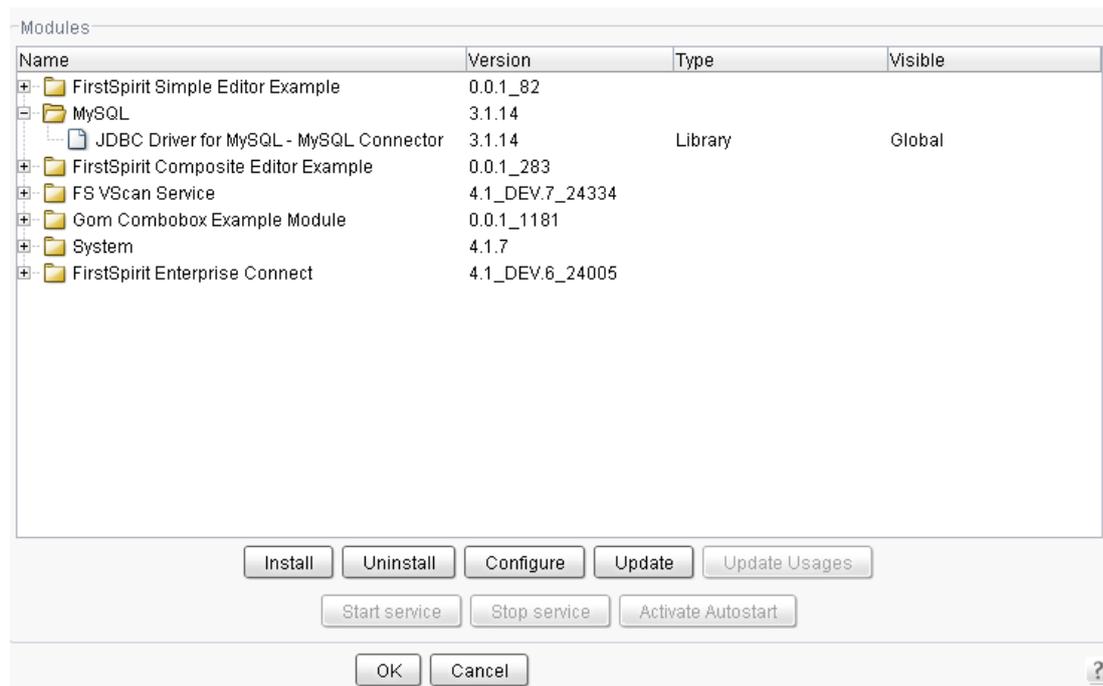


Abbildung 3-11: Installierte JDBC-Modul-Treiber-Bibliothek



Die in diesem Abschnitt beschriebene Variante sollte immer die bevorzugte Art der Implementierung sein.



3.17 Modul-Implementierung mit den Komponenten-Typen – PUBLIC, SERVICE, LIBRARY

Dieses Kapitel soll die Integration eines FirstSpirit Virenschanner-Moduls mittels der FirstSpirit Komponenten-Typen Service, Public und Library anhand einer Beispiel-Implementierung aufzeigen. Die voraussichtlich benötigte Infrastruktur für eine Virenschanner-Implementierung wurde mit dem VScan-Modul geschaffen, sowohl auf Modul-Ebene als auch im FirstSpirit-Server. Die initiale Proof-of-Concept-Implementierung nutzt hierbei einen lokal auf dem FirstSpirit-Server-System installierten Virenschanner (ClamAV / Linux). Dieser wird über einen ProcessBuilder angesprochen, was nur als Beispiel und zur Evaluierung des Konzepts dient. Ein ProcessBuilder ist im Produktionsbetrieb **nicht** geeignet für die Ausführung nativer Virenschanner-Anwendungen. Vielmehr sollte hier ein entfernter Service über Sockets, Pipes oder http (z.B. apache-commons mit Kapselung von ICAP) genutzt werden.

Die bisherige Umsetzung bietet ein Interface `ScanEngine`, welches von der konkreten Virenschanner-Engine implementiert werden muss. Daraus leitet sich der Komponenten-Typ PUBLIC für jede konkrete Virenschanner-Implementierung ab.

3.17.1 Modul-Komponenten und -Konfiguration

Das „VScan“-Modul beinhaltet mehrere Komponenten-Typen, wobei jede dieser Komponenten wiederum Ressourcen beinhaltet.

FS VScan Service	4.1_DEV.0_...		
VScanService	4.1_DEV.0_...	Dienst	Global
VScanFilterProxy	4.1_DEV.0_...	PUBLIC	Global
ClamAvEngine	4.1_DEV.0_...	PUBLIC	Global
libclamav	4.1_DEV.0_...	Bibliothek	Global

Abbildung 3-12: Komponenten-Typen des VScan-Moduls in der FirstSpirit Server- und Projektkonfiguration

Jede konkrete Implementierung einer Scanning-Engine enthält aus Modulsicht immer eine **PUBLIC**- und eine **LIBRARY**-Komponente. Wobei die PUBLIC-Komponente in Abbildung 3-12 ClamScanEngine.class das public Interface `ScanEngine` implementiert. Die Library ist das Archiv der Engine, welches weitere Klassen und Ressourcen enthält.

Die VScanService-Komponente ist konfigurierbar, d.h. für den Komponenten-Deskriptor, das Attribut `<configurable/>` muss mit der Klasse für die grafische Konfigurationsoberfläche definiert werden. Die Definition im Modul- bzw. im



Komponenten-Deskriptor stellt sich wie folgt dar (siehe auch Kapitel 3.9.1.3 Seite 44):

```
<configurable>de.espirit.firstspirit.opt.vscan.admin.gui.VscanServiceConfigPanel</configurable>
```

Die definierte Klasse muss wiederum das typisierte Interface Configuration implementieren:

```
public class VScanServiceConfigPanel implements  
de.espirit.firstspirit.module.Configuration<ServerEnvironment>
```

Die folgende Abbildung zeigt die Service-Konfigurationsoberfläche in der FirstSpirit Server- und Projektkonfiguration:

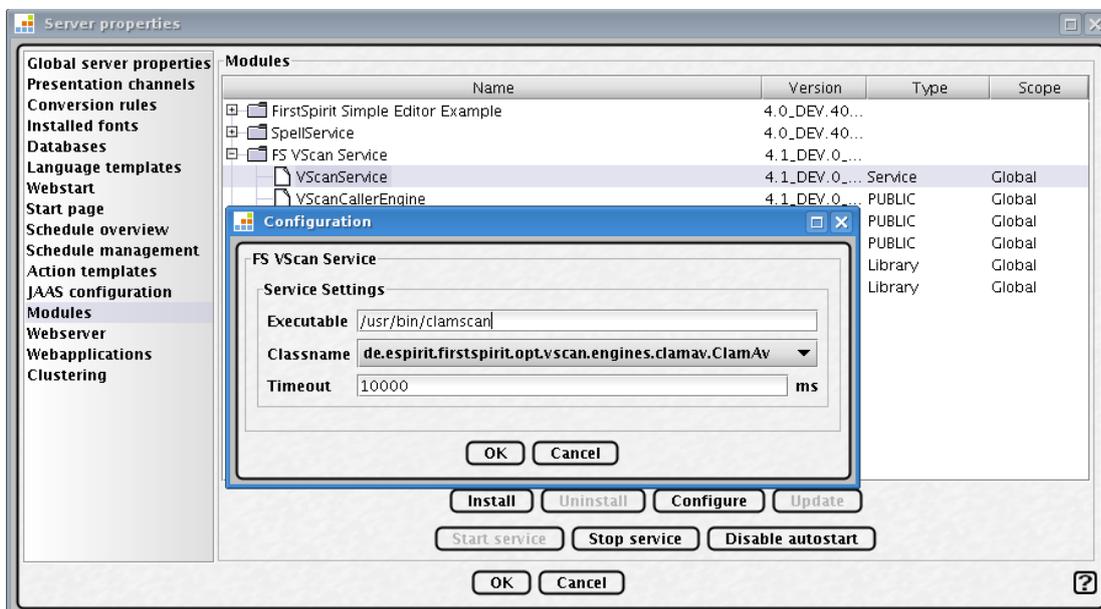


Abbildung 3-13: Service-Konfigurationsoberfläche in der FirstSpirit Server- und Projektkonfiguration

Die ComboBox **Classname** listet alle konkreten Scanning-Engines auf, die das ScanEngine Interface (HotSpot) implementieren.



3.17.2 Schematische Darstellung des Moduls innerhalb von FirstSpirit

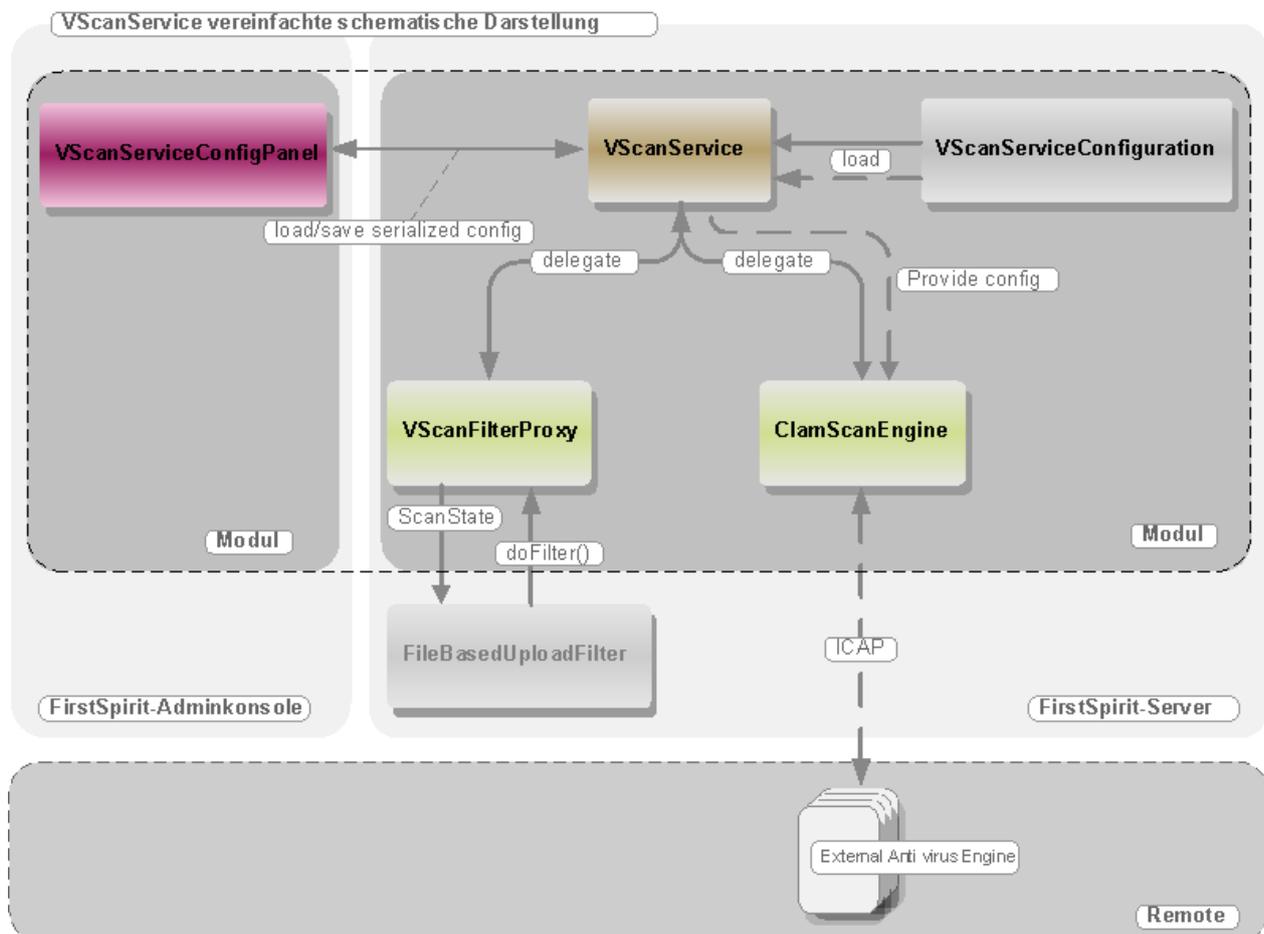


Abbildung 3-14: Kapselung von Modul-Komponenten innerhalb der FirstSpirit Server- und Projektkonfiguration sowie Kommunikationswege der Komponenten und Anbindung an die externe ICAP-Anwendung

3.17.3 Der VScan-Module-Deskriptor

Konkret heißt das für das VScan-Modul, es werden drei verschiedene Komponenten-Typen genutzt:

- Über den **service** Komponenten-Typ lässt sich der Virens Scanner-Service global (serverweit) starten, stoppen und konfigurieren.
- Der **library** Komponenten-Typ wird hier genutzt, um die verschiedenen Scanner-Engines dem System bekannt zu machen.
- Der **public** Komponenten-Typ implementiert das Interface ScanEngine des VScan-Moduls. Die Deskriptor-Datei „module.xml“ befindet sich im Wurzelverzeichnis des Modul-Sourcecodes. Generell besteht hier auch die



Möglichkeit, weitere Scanning-Engines als abhängiges Modul zu implementieren und nicht als weitere Komponente eines bestehenden Moduls. Wird ein neues Modul definiert, muss dieses auch wieder die Public- sowie die Library-Komponente enthalten, darüber hinaus muss eine Abhängigkeit zum VScan-Modul im Modul-Deskriptor (module.xml) definiert werden.

```
<dependencies>
  <depends>VScanService</depends>
</dependencies>
```

3.17.4 Vollständiger Modul-Deskriptor mit Drei Komponenten-Typen

```
1. <!DOCTYPE module SYSTEM "../server/module.dtd">
2. <module>
3.     <name>FS VScan Service</name>
4.     <version>@VERSION@</version>
5.     <description>Plugable Virus Scanning Service</description>
6.     <vendor>e-Spirit AG</vendor>
7.     <components>
8.         <service>
9.             <name>VScanService</name>
10.            <description>FirstSpirit Virus Scan
11.            Service</description>
12.            <class>de.espirit.firstspirit.opt.vscan.VScanServiceImpl</class>
13.            <configurable>de.espirit.firstspirit.opt.vscan.admin.gui.VScanServiceC
14.            onfigPanel</configurable>
15.            <resources>
16.                <resource name="libvscan">lib/fs-
17.                vscan.jar</resource>
18.                <resource>fs-vscan.conf</resource>
19.            </resources>
20.        </service>
21.        <public>
22.            <name>VScanFilterProxy</name>
23.            <description>The main engine which calls the
24.            specialized engine implementations.</description>
25.            <class>de.espirit.firstspirit.opt.vscan.VScanFilterProxy</class>
26.            <hidden>true</hidden>
27.        </public>
28.    </components>
29. </module>
```



```
24.         <depends>VScanService</depends>
25.     </dependencies>
26. </public>
27. <!-- Scanning Engines Listing -->
28. <public>
29.     <name>ClamAvEngine</name>
30.     <description>ClamAv core class implementing the
    AvEngine interface</description>
31. <class>de.espirit.firstspirit.opt.vscan.engines.clamav.ClamScanEngine<
/class>
32.     <dependencies>
33.         <depends>libclamav</depends>
34.         <depends>VScanCallerEngine</depends>
35.     </dependencies>
36. </public>
37. <library>
38.     <name>libclamav</name>
39.     <description>FS VScan Library containing the
    ClamAv virus scanning engine</description>
40.     <hidden>true</hidden>
41.     <resources>
42.         <resource name="libclamav">lib/engines/fs-
    clamav.jar</resource>
43.     </resources>
44.     <dependencies>
45.         <depends>ClamAvEngine</depends>
46.     </dependencies>
47. </library>
48. </components>
49. </module>
```

Listing 60: Drei Komponenten-Typen Modul-Deskriptor



3.17.5 Implementierung der SERVICE Basis-Komponente

Interface: VScanService

Package: de.espirit.firstspirit.opt.vscan

Definition der grundlegenden Service-Eigenschaften, wie z.B. Status-Codes, Laden der Konfiguration.

Interface 1: de.espirit.firstspirit.opt.vscan:VScanService

Klasse: VscanServiceImpl

Package: de.espirit.firstspirit.opt.vscan

Konkrete Implementierung der Service-Komponente. Start-, Stopp-Behandlung, Reagieren auf Ereignisse wie Installation, Aktualisierung, Deinstallation. Als Beispiel ist hier das Kopieren der Konfigurationsdatei `fs-vscan.conf` bei der Installation des Moduls

(`conf/modules/FS VScan Service.VScanService`) zu nennen.

public class VScanServiceImpl **implements** VScanService,
de.espirit.firstspirit.module.Service<VScanService>

Class 1: de.espirit.firstspirit.opt.vscan.VScanServiceImpl

```
1. package de.espirit.firstspirit.opt.vscan;
2.
3. import de.espirit.common.base.Logging;
4. import de.espirit.common.io.IoUtil;
5. import de.espirit.firstspirit.io.FileHandle;
6. import de.espirit.firstspirit.io.ServerConnection;
7.
8. import de.espirit.firstspirit.module.ServerEnvironment;
9. import de.espirit.firstspirit.module.Service;
10. import de.espirit.firstspirit.module.ServiceProxy;
11. import de.espirit.firstspirit.module.descriptor.ServiceDescriptor;
12. import org.jetbrains.annotations.NotNull;
13.
14. import java.io.ByteArrayInputStream;
15. import java.io.File;
16. import java.io.IOException;
17. import java.io.InputStream;
18. import java.io.OutputStream;
19. import java.io.UnsupportedEncodingException;
```



```
20. import java.util.ArrayList;
21. import java.util.Collections;
22. import java.util.List;
23. import java.util.Map;
24. import java.util.Map.Entry;
25.
26.
27. /**
28.  * $Date: 2008-06-02 17:25:13 +0200 (Mo, 02 Jun 2008) $
29.  *
30.  * @version $Revision: 22746 $
31.  */
32. public class VScanServiceImpl implements VScanService,
    Service<VScanService> {
33.
34.     private static final Class<?> LOGGER = VScanServiceImpl.class;
35.     public static final String MODULE_CONFIG_FILE = "fs-vscan.conf"; //
    NON-NLS
36.     public static final String MODULE_LOG_FILE = MODULE_NAME +
    ".log"; // NON-NLS
37.
38.     private ServerEnvironment _environment;
39.     private volatile boolean _running;
40.     private VScanServiceConfiguration _config;
41.     private FileHandle _configFile;
42.
43.
44.
45.
46.     public VScanServiceImpl() {
47.         _config = new VScanServiceConfiguration();
48.     }
49.
50.
51.
52.
53.     /** Starts the service. */
54.     public void start() {
55.         Logging.logInfo("Starting " + VScanService.MODULE_SERVICE_NAME
    + "...", LOGGER); // NON-NLS
56.         appendLog("Starting..."); // NON-NLS
57.         loadConfiguration();
58.         _running = true;
```



```
59.     }
60.
61.
62.
63.
64.     /** Stops the service. */
65.     public void stop() {
66.         Logging.logInfo("Shutting down " +
67. VScanService.MODULE_SERVICE_NAME + "...", LOGGER); // NON-NLS
68.         appendLog("Shutting down..."); // NON-NLS
69.         _running = false;
70.     }
71.
72.
73.
74.     /**
75.      * Returns whether the service is running.
76.      *
77.      * @return <code>>true</code> if the service is running.
78.      */
79.     public boolean isRunning() {
80.         return _running;
81.     }
82.
83.
84.
85.
86.     /**
87.      * Returns the service interface. Only methods of this interface
88.      are accessible, so <code>Service</code> instances must
89.      also implement this interface.
90.      *
91.      * @return service interface class.
92.      */
93.     public Class<? extends VScanService> getServiceInterface() {
94.         return VScanService.class;
95.     }
96.
97.
98.
99.     /**
```



```
100.      * A service proxy is an optional, client-side service
101.      * implementation. It will be instantiated, {@link
102.      * de.espirit.firstspirit.module.ServiceProxy#init(Object,
103.      * de.espirit.firstspirit.access.Connection) initialized} and
104.      * returned by {@link
105.      * de.espirit.firstspirit.access.Connection#getService(String)}. The proxy
106.      * class must have a no-arg
107.      * constructor and must implement the {@link
108.      * de.espirit.firstspirit.module.ServiceProxy} interface, but has not to
109.      * implement the {@link #getServiceInterface() service-interface}
110.      * itself.
111.      *
112.      * @return service proxy class or <code>null</code> if no proxy is
113.      * provided.
114.      */
115.      public Class<? extends ServiceProxy> getProxyClass() {
116.          return null;
117.      }
118.
119.
120.
121.
122.
123.
124.
125.
126.
127.
128.
129.
130.
131.
132.
133.
134.
135.
136.
137.
138.
139.
140.
141.
142.
143.
144.
145.
146.
147.
148.
149.
150.
151.
152.
153.
154.
155.
156.
157.
158.
159.
160.
161.
162.
163.
164.
165.
166.
167.
168.
169.
170.
171.
172.
173.
174.
175.
176.
177.
178.
179.
180.
181.
182.
183.
184.
185.
186.
187.
188.
189.
190.
191.
192.
193.
194.
195.
196.
197.
198.
199.
200.
201.
202.
203.
204.
205.
206.
207.
208.
209.
210.
211.
212.
213.
214.
215.
216.
217.
218.
219.
220.
221.
222.
223.
224.
225.
226.
227.
228.
229.
230.
231.
232.
233.
234.
235.
236.
237.
238.
239.
240.
241.
242.
243.
244.
245.
246.
247.
248.
249.
250.
251.
252.
253.
254.
255.
256.
257.
258.
259.
260.
261.
262.
263.
264.
265.
266.
267.
268.
269.
270.
271.
272.
273.
274.
275.
276.
277.
278.
279.
280.
281.
282.
283.
284.
285.
286.
287.
288.
289.
290.
291.
292.
293.
294.
295.
296.
297.
298.
299.
300.
301.
302.
303.
304.
305.
306.
307.
308.
309.
310.
311.
312.
313.
314.
315.
316.
317.
318.
319.
320.
321.
322.
323.
324.
325.
326.
327.
328.
329.
330.
331.
332.
333.
334.
335.
336.
337.
338.
339.
340.
341.
342.
343.
344.
345.
346.
347.
348.
349.
350.
351.
352.
353.
354.
355.
356.
357.
358.
359.
360.
361.
362.
363.
364.
365.
366.
367.
368.
369.
370.
371.
372.
373.
374.
375.
376.
377.
378.
379.
380.
381.
382.
383.
384.
385.
386.
387.
388.
389.
390.
391.
392.
393.
394.
395.
396.
397.
398.
399.
400.
401.
402.
403.
404.
405.
406.
407.
408.
409.
410.
411.
412.
413.
414.
415.
416.
417.
418.
419.
420.
421.
422.
423.
424.
425.
426.
427.
428.
429.
430.
431.
432.
433.
434.
435.
436.
437.
438.
439.
440.
441.
442.
443.
444.
445.
446.
447.
448.
449.
450.
451.
452.
453.
454.
455.
456.
457.
458.
459.
460.
461.
462.
463.
464.
465.
466.
467.
468.
469.
470.
471.
472.
473.
474.
475.
476.
477.
478.
479.
480.
481.
482.
483.
484.
485.
486.
487.
488.
489.
490.
491.
492.
493.
494.
495.
496.
497.
498.
499.
500.
501.
502.
503.
504.
505.
506.
507.
508.
509.
510.
511.
512.
513.
514.
515.
516.
517.
518.
519.
520.
521.
522.
523.
524.
525.
526.
527.
528.
529.
530.
531.
532.
533.
534.
535.
536.
537.
538.
539.
540.
541.
542.
543.
544.
545.
546.
547.
548.
549.
550.
551.
552.
553.
554.
555.
556.
557.
558.
559.
560.
561.
562.
563.
564.
565.
566.
567.
568.
569.
570.
571.
572.
573.
574.
575.
576.
577.
578.
579.
580.
581.
582.
583.
584.
585.
586.
587.
588.
589.
590.
591.
592.
593.
594.
595.
596.
597.
598.
599.
600.
601.
602.
603.
604.
605.
606.
607.
608.
609.
610.
611.
612.
613.
614.
615.
616.
617.
618.
619.
620.
621.
622.
623.
624.
625.
626.
627.
628.
629.
630.
631.
632.
633.
634.
635.
636.
637.
638.
639.
640.
641.
642.
643.
644.
645.
646.
647.
648.
649.
650.
651.
652.
653.
654.
655.
656.
657.
658.
659.
660.
661.
662.
663.
664.
665.
666.
667.
668.
669.
670.
671.
672.
673.
674.
675.
676.
677.
678.
679.
680.
681.
682.
683.
684.
685.
686.
687.
688.
689.
690.
691.
692.
693.
694.
695.
696.
697.
698.
699.
700.
701.
702.
703.
704.
705.
706.
707.
708.
709.
710.
711.
712.
713.
714.
715.
716.
717.
718.
719.
720.
721.
722.
723.
724.
725.
726.
727.
728.
729.
730.
731.
732.
733.
734.
735.
736.
737.
738.
739.
740.
741.
742.
743.
744.
745.
746.
747.
748.
749.
750.
751.
752.
753.
754.
755.
756.
757.
758.
759.
760.
761.
762.
763.
764.
765.
766.
767.
768.
769.
770.
771.
772.
773.
774.
775.
776.
777.
778.
779.
780.
781.
782.
783.
784.
785.
786.
787.
788.
789.
790.
791.
792.
793.
794.
795.
796.
797.
798.
799.
800.
801.
802.
803.
804.
805.
806.
807.
808.
809.
810.
811.
812.
813.
814.
815.
816.
817.
818.
819.
820.
821.
822.
823.
824.
825.
826.
827.
828.
829.
830.
831.
832.
833.
834.
835.
836.
837.
838.
839.
840.
841.
842.
843.
844.
845.
846.
847.
848.
849.
850.
851.
852.
853.
854.
855.
856.
857.
858.
859.
860.
861.
862.
863.
864.
865.
866.
867.
868.
869.
870.
871.
872.
873.
874.
875.
876.
877.
878.
879.
880.
881.
882.
883.
884.
885.
886.
887.
888.
889.
890.
891.
892.
893.
894.
895.
896.
897.
898.
899.
900.
901.
902.
903.
904.
905.
906.
907.
908.
909.
910.
911.
912.
913.
914.
915.
916.
917.
918.
919.
920.
921.
922.
923.
924.
925.
926.
927.
928.
929.
930.
931.
932.
933.
934.
935.
936.
937.
938.
939.
940.
941.
942.
943.
944.
945.
946.
947.
948.
949.
950.
951.
952.
953.
954.
955.
956.
957.
958.
959.
960.
961.
962.
963.
964.
965.
966.
967.
968.
969.
970.
971.
972.
973.
974.
975.
976.
977.
978.
979.
980.
981.
982.
983.
984.
985.
986.
987.
988.
989.
990.
991.
992.
993.
994.
995.
996.
997.
998.
999.
1000.
```



```
130.     /** Event method: called if Component was successfully installed  
131.     (not updated!) */  
132.     public void installed() {  
133.         copyConfigFile(getConfigFileName()); // NON-NLS  
134.         Logging.logDebug(getName() + " installed...", LOGGER); // NON-  
135.         NLS  
136.         appendLog("installed"); // NON-NLS  
137.     }  
138.  
139.  
140.     /** Event method: called if Component was in uninstalling  
141.     procedure */  
142.     public void uninstalling() {  
143.         Logging.logDebug(getName() + " uninstalling...", LOGGER); //  
144.         NON-NLS  
145.     }  
146.  
147.  
148.     /**  
149.     * Event method: called if Component was completely updated  
150.     *  
151.     * @param oldVersionString old version, before component was  
152.     updated  
153.     */  
154.     public void updated(final String oldVersionString) {  
155.         // e.g. merge, diff, bkup the config file or do not overwrite  
156.         copyConfigFile(getConfigFileName()); // NON-NLS  
157.         Logging.logDebug(getName() + " updated...", LOGGER); // NON-NLS  
158.         appendLog("updated"); // NON-NLS  
159.     }  
160.  
161.  
162.  
163.     /**  
164.     * Copy a config file from the package dir to module config dir ->  
165.     conf/modules/$module_name$. $module_service_name$  
166.     *  
167.     * @param file the module config file name string
```



```
167.     */
168.     private void copyConfigFile(final String file) {
169.         try {
170.             final FileHandle configFile =
171.                 _environment.getConfDir().obtain(file);
172.             if (configFile.exists()) {
173.                 Logging.logDebug("config file " + configFile.getPath()
174. + " exists, do not overwrite", LOGGER); // NON-NLS
175.             } else {
176.                 final InputStream is =
177.                     getClass().getResourceAsStream(file);
178.                 if (is != null) {
179.                     try {
180.                         configFile.save(is);
181.                     } finally {
182.                         IoUtil.saveClose(is);
183.                     }
184.                 }
185.             } catch (final IOException e) {
186.                 Logging.logError("Error saving file - for " + getName() +
187. + " compoment", e, LOGGER); // NON-NLS
188.             }
189.         }
190.     }
191.
192.     private void appendLog(final String message) {
193.         final StringBuilder sb = new StringBuilder();
194.         sb.append(Logging.getDateString());
195.         sb.append('\t');
196.         sb.append(message);
197.         sb.append('\n');
198.
199.         InputStream is = null;
200.         try {
201.             //final FileHandle handle =
202.                 _environment.getConfDir().obtain(MODULE_LOG_FILE);
203.             final FileHandle handle =
204.                 _environment.getLogDir().obtain(MODULE_LOG_FILE);
205.             is = new ByteArrayInputStream(sb.toString()).getBytes("UTF-
```



```
8"));// NON-NLS
204.         handle.append(is);
205.     } catch (final UnsupportedOperationException e) {
206.         throw new IllegalStateException(e);// should not happen
207.     } catch (final IOException e) {
208.         Logging.logError("Couldn't obtain file: " +
MODULE_LOG_FILE, e, LOGGER);// NON-NLS
209.     } finally {
210.         IoUtil.saveClose(is);
211.     }
212. }

213. -----
214.
215.
216.
217.
218.     @SuppressWarnings({"UnusedCatchParameter"})
219.     private ScanEngine getScanEngine(@NotNull final String className)
throws ClassNotFoundException {
220.         // replace single call to loadAvEngine by a pre loaded engine
pool on service start
221.         try {
222.             final Class<?> clazz =
getClass().getClassLoader().loadClass(className);
223.             Logging.logInfo("Loading Anti Virus Scanning Engine: " +
className + " ...", LOGGER);// NON-NLS
224.             if (!ScanEngine.class.isAssignableFrom(clazz)) {
225.                 throw new IllegalStateException(clazz + " does not
implement " + ScanEngine.class.getName());
226.             }
227.             final ScanEngine result = (ScanEngine)
clazz.newInstance();
228.             result.init(_config);
229.             if (result.isThreadSafe()) {
230.                 // todo: pooling
231.             }
232.             return result;
233.         } catch (InstantiationException e) {
234.             throw new InstantiationError("Could not instantiate virus
scan engine " + className);
235.         } catch (IllegalAccessException e) {
236.             throw new IllegalStateException("Access denied - class: "
+ className);
```



```
237.         }
238.     }
239.
240.
241.
242.
243.     public void scanFile(@NotNull final File tempFile) throws
ClassNotFoundException, UploadRejectedException {
244.         try {
245.             final ScanEngine scanEngine =
getScanEngine(_config.getClassName());
246.             appendLog("scanning " + tempFile + " with " +
scanEngine.getName()); // NON-NLS
247.             scanEngine.scanFile(tempFile);
248.             appendLog("file " + tempFile + " ok"); // NON-NLS
249.         } catch (final UploadRejectedException e) {
250.             appendLog("file " + tempFile + " not ok - " +
e.getMessage()); // NON-NLS
251.             throw e;
252.         } catch (final ClassNotFoundException e) {
253.             Logging.logError("Loading of virus scanning engine failed:
" + _config.getClassName(), e, LOGGER); // NON-NLS
254.             throw new ClassNotFoundException("Loading of virus
scanning engine failed: " + _config.getClassName(), e);
255.         }
256.     }
257.
258.
259.     // VScanServiceConfiguration
260.
261.
262.
263.
264.     public void loadConfiguration() {
265.         InputStream is = null;
266.
267.         _config.setEnvironment(_environment);
268.
269.         try {
270.             _configFile =
_environment.getConfDir().obtain(MODULE_CONFIG_FILE);
271.             is = _configFile.load();
272.         }
```



```
273.         if (_configFile.isFile()) {
274.             _config.getServiceProperties().load(is);
275.             _config.init();
276.             _config.setAvailableEngines(getAvailableEngines());
277.         }
278.     } catch (IOException e) {
279.         Logging.logDebug("Couldn't load config file - " +
280.             _configFile.getPath(), e, LOGGER); // NON-NLS
281.     } finally {
282.         try {
283.             if (is != null) {
284.                 is.close();
285.             }
286.         } catch (IOException e) {
287.             Logging.logDebug("Couldn't close config file - " +
288.                 _configFile.getPath(), e, LOGGER); // NON-NLS
289.         }
290.     }
291.
292.
293.
294.     public void saveConfiguration() {
295.         OutputStream os = null;
296.
297.         try {
298.             if (_configFile.exists() && _configFile.isFile()) {
299.                 IoUtil.copyFile(_configFile.getPath(),
300.                     _configFile.getPath() + ".1");
301.             }
302.
303.             os = _configFile.getOutputStream();
304.
305.             _config.getServiceProperties().setProperty("fsm.vscan.engine.executable",
306.                 _config.getExecutable().toString());
307.             Logging.logDebug("Setting property -
308.                 fsm.vscan.engine.executable=" + _config.getExecutable().toString(),
309.                 LOGGER); // NON-NLS
310.
311.             _config.getServiceProperties().setProperty("fsm.vscan.engine.class",
```



```
_config.getClassName());
308.         Logging.logDebug("Setting property -
    fsm.vscan.engine.class=" + _config.getClassName(), LOGGER); // NON-NLS
309.
310.
    _config.getServiceProperties().setProperty("fsm.vscan.engine.timeout",
    String.valueOf(_config.getTimeout()));
311.         Logging.logDebug("Setting property -
    fsm.vscan.engine.timeout=" + String.valueOf(_config.getTimeout()),
    LOGGER); // NON-NLS
312.
313.         _config.getServiceProperties().store(os, "Last
    modified"); // NON-NLS
314.
315.     } catch (IOException e) {
316.         Logging.logDebug("Couldn't open config file: " +
    _configFile.getPath(), e, LOGGER); // NON-NLS
317.     } finally {
318.         try {
319.             if (os != null) {
320.                 os.close();
321.             }
322.         } catch (IOException e) {
323.             Logging.logDebug("Closing of config file output stream
    failed.", e, LOGGER); // NON-NLS
324.         }
325.     }
326. }
327.
328.
329.
330.
331.     /** {@inheritDoc} */
332.     @NotNull
333.     public String getName() {
334.         return MODULE_NAME;
335.     }
336.
337.
338.
339.
340.     /**
341.      * Get the module configuration file name
```



```
342.     *
343.     * @return the config file name string
344.     */
345.     @NotNull
346.     private static String getConfigFileName() {
347.         return MODULE_CONFIG_FILE;
348.     }
349.
350.
351.
352.
353.     /** {@inheritDoc} */
354.     public boolean isEnabled() {
355.         return _running;
356.     }
357.
358.
359.
360.
361.     /** {@inheritDoc} */
362.     public VScanServiceConfiguration getServiceConfiguration() {
363.         loadConfiguration();
364.         return _config;
365.     }
366.
367.
368.
369.
370.     public void setServiceConfiguration(final
371. VScanServiceConfiguration config) {
372.         _config = config; // todo: clear pool
373.         saveConfiguration();
374.     }
375. }
```

Listing 61: de.espirit.firstspirit.opt.vscan.VScanServiceImpl



Klasse: VScanServiceConfiguration

Package: de.espirit.firstspirit.opt.vscan

Modell für die notwendigen Konfigurations-Einstellungen des Services. Getter und Setter zu jeder Konfigurationseigenschaft. Lädt bei Initialisierung die persistenten Einstellungen aus der `fs-vscan.conf` Datei. Besitzt eine `save()`-Methode, um die geladenen oder in der Konfigurationsoberfläche geänderten Einstellungen in die Konfigurationsdatei zu schreiben. Jedes Key/Value-Paar in der `fs-vscan.conf` entspricht hier einer Member-Variable der Klasse und benötigt hierfür jeweils einen Getter und Setter. Die `init()`-Methode muss ggf. um die weiteren benötigten Konfigurationsoptionen erweitert werden. Laden und Speichern der Konfigurationsoptionen erfolgt über `VScanServiceImpl#saveConfiguration()` und `VScanServiceImpl#loadConfiguration`. Die Konfigurationsklasse implementiert hier das Interface `Serializable`, das diese vom serverseitigen Service zur FirstSpirit Server- und Projektkonfiguration übertragen wird. Der Service kontrolliert den Lifecycle der Konfiguration.

public class VScanServiceConfiguration **implements** Serializable

Class 2: de.espirit.firstspirit.opt.vscan.VScanServiceConfiguration



Klasse: VscanServiceConfigPanel

Package: de.espirit.firstspirit.opt.vscan

Die Service-Konfigurationsoberfläche, zu erreichen über die FirstSpirit Server- und Projekteigenschaften. Alle notwendigen Felder der VscanServiceConfiguration können hier geladen werden – die Methode `getGui()` initialisiert die Konfigurationsoberfläche, wobei die Methode `hasGui()` hartcodiert `true` zurückliefern muss.

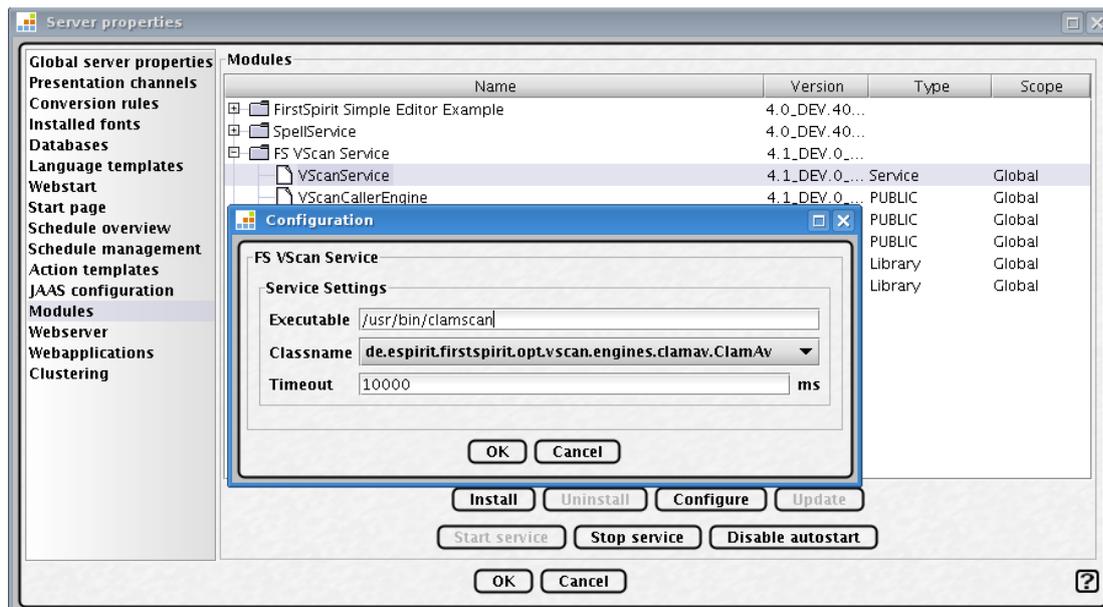


Abbildung 3-15: VscanServiceConfigPanel – Die Modul-Konfigurationsoberfläche

Class 3: de.espirit.firstspirit.opt.vscan.VScanServiceConfigPanel

```

1. package de.espirit.firstspirit.opt.vscan.admin.gui;
2.
3. import de.espirit.firstspirit.access.ServiceNotFoundException;
4. import de.espirit.firstspirit.common.gui.CMSDialog;
5. import de.espirit.firstspirit.io.ServerConnection;
6.
7. import de.espirit.firstspirit.module.Configuration;
8. import de.espirit.firstspirit.module.ServerEnvironment;
9. import de.espirit.firstspirit.opt.vscan.VScanService;
10. import de.espirit.firstspirit.opt.vscan.VScanServiceConfiguration;
11. import de.espirit.firstspirit.opt.vscan.resources.ModuleResources;
12. import de.espirit.firstspirit.server.ManagerNotFoundException;
13. import info.clearthought.layout.TableLayout;

```



```
14.
15. import javax.swing.BorderFactory;
16. import javax.swing.JComboBox;
17. import javax.swing.JComponent;
18. import javax.swing.JLabel;
19. import javax.swing.JPanel;
20. import javax.swing.JTextField;
21. import java.awt.Frame;
22. import java.net.URI;
23. import java.util.List;
24. import java.util.Set;
25.
26.
27. /**
28.  * $Date: 2008-06-02 17:25:13 +0200 (Mo, 02 Jun 2008) $
29.  *
30.  * @version $Revision: 22746 $
31.  */
32. public class VScanServiceConfigPanel implements
Configuration<ServerEnvironment> {
33.
34.     private VScanServiceConfiguration _config;
35.     private ServerEnvironment _env;
36.     private JPanel _component;
37.     private JTextField _engineExecutableField;
38.     private JTextField _engineTimeoutField;
39.     private JComboBox _enginesClassList;
40.     private List<String> _availableEngines;
41.
42.
43.
44.
45.     public VScanServiceConfigPanel() {
46.     }
47.
48.
49.
50.
51.     /**
52.      * Returns true if this component has a gui to show
and change it's configuration.
53.      *
54.      * @return true if this component has a gui to show
```



```
and change it's configuration.
55.     */
56.     public boolean hasGui() {
57.         return true;
58.     }
59.
60.
61.
62.
63.     /**
64.      * Returns the configuration gui.
65.      *
66.      * @param masterFrame basic frame component, for creating new
        windows
67.      * @return configuration gui or null.
68.      */
69.     public JComponent getGui(final Frame masterFrame) {
70.         if (_component == null) {
71.
72.             final double border = 5;
73.             final double rowsGap = 5;
74.             final double[][] size = {{border, TableLayout.FILL,
        border}, {border, TableLayout.PREFERRED, rowsGap, TableLayout.PREFERRED,
        rowsGap, TableLayout.PREFERRED, border}};
75.             final TableLayout tbl = new TableLayout(size);
76.
77.             final JPanel panel = new JPanel();
78.             panel.setOpaque(false);
79.
        panel.setBorder(BorderFactory.createTitledBorder(VScanService.MODULE_NAME))
80.
81.             ;
82.             panel.setLayout(tbl);
83.
84.             panel.add(getEnginePanel(), "1, 1, 1, 1");
85.
86.             _component = (JPanel) masterFrame.add(panel);
87.             //_component = panel;
88.
89.             clearGuiValues();
90.             initGuiValues();
91.         }
92.
93.         return _component;
94.     }
95. }
```



```
92.     }
93.
94.
95.
96.
97.     private JComponent getEnginePanel() {
98.         final double border = 5;
99.         final double rowsGap = 5;
100.        final double colsGap = 5;
101.        final double[][] size = {{border, TableLayout.PREFERRED,
102.        colsGap, TableLayout.FILL, colsGap, TableLayout.MINIMUM, border}, {border,
103.        TableLayout.PREFERRED, rowsGap, TableLayout.PREFERRED, rowsGap,
104.        TableLayout.PREFERRED, border}};
105.        final TableLayout tbl = new TableLayout(size);
106.
107.        final JPanel enginePanel = new JPanel();
108.        enginePanel.setOpaque(false);
109.
110.        enginePanel.setBorder(BorderFactory.createTitledBorder(ModuleResources.getS
111.        tring("fs-
112.        resource.module.vscan.admin.gui.VScanServiceConfigPanel_EngineSetup")));
113.        enginePanel.setLayout(tbl);
114.
115.        enginePanel.add(new JLabel(ModuleResources.getString("fs-
116.        resource.module.vscan.admin.gui.VScanServiceConfigPanel_EngineExecutable")
117.        , "1, 1, 1, 1"));
118.        //enginePanel.add(new
119.        JLabel(ModuleResources.loadIconResource("de.espirit.firstspirit.opt.vscan.r
120.        esources.icons", "core.png")), "1, 1, 1, 1");// NON-NLS
121.        //enginePanel.add(new
122.        JLabel(ModuleResources.loadIconResource("core.png")), "1, 1, 1, 1");// NON-
123.        NLS
124.        _engineExecutableField = new JTextField();
125.        enginePanel.add(_engineExecutableField, "3, 1, 5, 1");
126.
127.
128.
129.        _enginesClassList = new JComboBox(loadEnginesList());
130.        enginePanel.add(new JLabel(ModuleResources.getString("fs-
131.        resource.module.vscan.admin.gui.VScanServiceConfigPanel_EngineClassname")),
132.        "1, 3, 1, 3");
133.        enginePanel.add(_enginesClassList, "3, 3, 5, 3");
134.
135.
136.
137.
138.
139.
140.
```



```
121.         enginePanel.add(new JLabel(ModuleResources.getString("fs-
resource.module.vscan.admin.gui.VScanServiceConfigPanel_EngineTimeout")),
"1, 5, 1, 5");
122.         _engineTimeoutField = new JTextField();
123.         enginePanel.add(_engineTimeoutField, "3, 5, 3, 5");
124.         enginePanel.add(new JLabel("ms"), "5, 5, 5, 5");
125.
126.         return enginePanel;
127.     }
128.
129.
130.
131.
132.     /**
133.      * Loads the current configuration from appropriate {@link
de.espirit.firstspirit.module.ServerEnvironment#getConfDir()
134.      * conf directory}.
135.      */
136.     @SuppressWarnings({"UnusedCatchParameter"})
137.     public void load() {
138.         VScanService service = null;
139.
140.         try {
141.             service = (VScanService)
_env.getConnection().getService(VScanService.MODULE_SERVICE_NAME);
142.             if (!service.isEnabled()) {
143.                 showServiceStartDialog();
144.             }
145.         } catch (IllegalStateException e) {
146.             showServiceStartDialog();
147.         } catch (ServiceNotFoundException e) {
148.             showServiceStartDialog();
149.         } catch (ManagerNotFoundException e) {
150.             showServiceStartDialog();
151.         }
152.
153.         if(service == null) {
154.             service = (VScanService)
_env.getConnection().getService(VScanService.MODULE_SERVICE_NAME);
155.         }
156.
157.         _config = service.getServiceConfiguration();
158.         _config.setEnvironment(_env);
```



```
159.     }
160.
161.
162.
163.
164.
165.     /** Stores the current configuration. */
166.     public void store() {
167.         final VScanService service = (VScanService)
168.         _env.getConnection().getService(VScanService.MODULE_SERVICE_NAME);
169.
170.         _config.setExecutable(URI.create(_engineExecutableField.getText().trim()));
171.         _config.setClassName(_enginesClassList.getSelectedItem().toString());
172.         _config.setTimeout(Long.valueOf(_engineTimeoutField.getText().trim()));
173.         //_config.save();
174.         service.setServiceConfiguration(_config);
175.     }
176.
177.
178.
179.     private void clearGuiValues() {
180.         _engineExecutableField.setText("");
181.         _engineTimeoutField.setText("");
182.     }
183.
184.
185.
186.
187.     private void initGuiValues() {
188.
189.         _engineExecutableField.setText(_config.getExecutable().toString().trim());
190.         _engineTimeoutField.setText(String.valueOf(_config.getTimeout()).trim());
191.
192.         for (final Object engineClasses : _availableEngines) {
193.             if
194.             (_config.getClassName().equals(engineClasses.toString())) {
```



```
_enginesClassList.setSelectedItem(engineClasses.toString());
195.         }
196.     }
197. }
198.
199.
200.
201.
202.     private Object[] loadEnginesList() {
203.         _availableEngines = _config.getAvailableEngines();
204.
205.         if (_availableEngines.isEmpty()) {
206.             _availableEngines.add(ModuleResources.getString("fs-
resource.module.vscan.admin.gui.VScanServiceConfigPanel_ErrorNoEngineAvaila
ble"));
207.         }
208.
209.         return _availableEngines.toArray();
210.     }
211.
212.
213.
214.
215.     /**
216.      * Returns all parameter names.
217.      *
218.      * @return all parameter names.
219.      */
220.     public Set<String> getParameterNames() {
221.         return _config.getParameterNames();
222.     }
223.
224.
225.
226.
227.     /**
228.      * Returns a specific parameter or null if it's not
available.
229.      *
230.      * @param name parameter name.
231.      * @return parameter or null.
232.      */
233.     public String getParameter(final String name) {
```



```
234.         return _config.getParameter(name);
235.     }
236.
237.
238.
239.
240.     /**
241.      * Initializes this component with the given {@link
242.      * de.espirit.firstspirit.module.ServerEnvironment environment}. This
243.      * method is called before the instance is used.
244.      *
245.      * @param moduleName    module name
246.      * @param componentName component name
247.      * @param env            useful environment information for this
248.      *                       component.
249.      */
250.     public void init(final String moduleName, final String
251.     componentName, final ServerEnvironment env) {
252.         _env = env;
253.     }
254.
255.     /**
256.      * Returns ComponentEnvironment
257.      *
258.      * @return ComponentEnvironment
259.      */
260.     public ServerEnvironment getEnvironment() {
261.         return _env;
262.     }
263. }
```

Listing 62: de.espirit.firstspirit.opt.vscan.VScanServiceConfigPanel

Interface: ScanEngine

Package: de.espirit.firstspirit.opt.vscan

Definition der grundlegenden Methoden, die von einer VirusScan-Engine zur Verfügung gestellt werden müssen (siehe auch ClamScanEngine.java und ScanEngine#JavaDoc), damit diese ohne weiteres in die bestehende Infrastruktur eingebunden werden kann.

Interface 2: de.espirit.firstspirit.opt.vscan.ScanEngine



3.17.6 Die PUBLIC-Komponente des Moduls

Klasse: VScanFilterProxy

Package: de.espirit.firstspirit.opt.vscan

Erweitert die Klasse `FileBasedUploadFilter`, welche wiederum `UploadFilter` implementiert. Daraus ergibt sich der schon zuvor angesprochene Komponenten-Typ `Public` (siehe auch Kapitel 2.9.1.7 Seite 23).

Alle Aufrufe der Klasse werden an `VscanServiceImpl` delegiert.

Class 4: de.espirit.firstspirit.opt.vscan.VScanFilterProxy

```
1. package de.espirit.firstspirit.opt.vscan;
2.
3. import de.espirit.common.base.Logging;
4. import
   de.espirit.firstspirit.server.mediamanagement.FileBasedUploadFilter;
5.
6. import java.io.File;
7. import java.io.IOException;
8.
9.
10. /**
11.  * $Date: 2008-05-26 13:52:11 +0200 (Mo, 26 Mai 2008) $
12.  *
13.  * @version $Revision: 22535 $
14.  */
15. public class VScanFilterProxy extends FileBasedUploadFilter {
16.
17.     public static final String ENGINE_NAME = "VScanFilterProxy"; //
   NON-NLS
18.     public static final Class<?> LOGGER = VScanFilterProxy.class;
19.
20.
21.
22.
23.     public VScanFilterProxy() {
24.         // no arg constructor needed, called from super class
25.     }
26.
27.
28.
29.
```



```
30.     @Override
31.     public void init() {
32.         if (Logging.isDebugEnabled(LOGGER)) {
33.             Logging.logDebug("VScanFilterProxy.init() - ", LOGGER);//
NON-NLS
34.         }
35.     }
36.
37.
38.     // -- get VScanService, get VScanServiceConfiguration -- //
39.     -----
40.
41.
42.
43.     /**
44.      * Get the VScanService by the serviceLocator
45.      *
46.      * @return the VScanService
47.      */
48.     private VScanService getVScanService() {
49.         return (VScanService)
getServiceLocator().getService(VScanService.MODULE_SERVICE_NAME);
50.     }
51.
52.
53.     // -- Filter, Execute, Grant or Reject -- //
54.     -----
55.
56.
57.
58.     /** {@inheritDoc} */
59.     @Override
60.     public void doFilter(final File tempFile) throws IOException {
61.         try {
62.             getVScanService().scanFile(tempFile);
63.         } catch (ClassNotFoundException e) {
64.             Logging.logInfo("Loading of virus scanning engine
failed!", e, LOGGER);// NON-NLS
65.             throw new IOException("Loading of virus scanning engine
failed!");
66.         }
67.     }
68.
```



```
69.     }
```

Listing 63: de.espirit.firstspirit.opt.vscan.VScanFilterProxy

3.17.7 Die LIBRARY-Komponente(n) des Moduls

Klasse: ClamScanEngine**Package:** de.espirit.firstspirit.opt.vscan.engines.clamav.ClamScanEngine

Die konkrete Implementierung des `ScanEngine`-Interfaces. Diese Klasse setzt alle Stati, enthält die Logik zur Kommunikation mit der externen Scan-Engine und extrahiert aus dem Reply die nötigen Informationen, um einen eindeutigen Status zu setzen (`throw new UploadRejectedException`). Anstelle dieser Implementierung der `ClamScanEngine` (`/usr/bin/clamscan`) kann man sich bspw. auch eine HTTP/ICAP-Implementierung vorstellen.

Class 5: de.espirit.firstspirit.opt.vscan.engines.clamav.ClamScanEngine



Prinzipiell muss für die Einbindung weiterer Scanning-Engines nur die nötige(n) Klasse(n) für die spezielle AntiVirus-Anwendungen implementiert werden. Die `ClamScanEngine`-Klasse soll zeigen, wie es z.B. auch möglich wäre, eine ICAP-Umsetzung zu schaffen, die mittels ICAP-Client mit einer entfernten ICAP-AntiVirus-Anwendung (Symantec, Kaspersky etc.) kommuniziert.

Hierzu ist eine Anpassung anderer Klassen oder Interfaces nicht notwendig.

z.B.:

- `de.espirit.firstspirit.opt.vscan.engines.icap.IcapClient`



Eine Implementierung, welche die FirstSpirit UploadFilter-API nutzt, muss eine konkrete `UploadRejectedException` werfen, falls ein Medium nicht den Restriktionen entspricht oder im Falle der ScanEngine ein Virus gefunden wurde.

```
1.     package de.espirit.firstspirit.opt.vscan.engines.clamav;
2.
3.     import de.espirit.common.base.Logging;
4.     import de.espirit.firstspirit.access.AccessUtil;
5.     import de.espirit.firstspirit.opt.vscan.ScanEngine;
6.     import
de.espirit.firstspirit.access.store.mediastore.UploadRejectedException;
7.     import de.espirit.firstspirit.opt.vscan.VScanService;
8.     import de.espirit.firstspirit.opt.vscan.VScanServiceConfiguration;
```



```
9.
10. import java.io.File;
11. import java.io.IOException;
12. import java.io.StringWriter;
13. import java.net.URI;
14.
15.
16. /**
17.  * $Date: 2008-05-26 13:52:11 +0200 (Mo, 26 Mai 2008) $
18.  *
19.  * @version $Revision: 22535 $
20.  */
21. public class ClamScanEngine implements ScanEngine {
22.
23.     public static final Class<?> LOGGER = ClamScanEngine.class;
24.     public static final String ENGINE_NAME = "ClamAv"; // NON-NLS
25.
26.     private URI _executable;
27.
28.
29.
30.
31.     public ClamScanEngine() {
32.         Logging.logDebug("ClamScanEngine created", LOGGER); // NON-NLS
33.     }
34.
35.
36.
37.
38.     public String getName() {
39.         return ENGINE_NAME;
40.     }
41.
42.
43.
44.
45.     public void init(final VScanServiceConfiguration configuration) {
46.         _executable = configuration.getExecutable();
47.         Logging.logDebug("Executable set to: " + _executable,
48.             LOGGER); // NON-NLS
49.         if (! executableExists()) {
50.             Logging.logError(VScanService.MODULE_NAME + ": " +
51.                 getName() + " executable '" + getExecutable() + "' not found.", LOGGER); //
```



```
NON-NLS
50.         throw new IllegalStateException(VScanService.MODULE_NAME +
51.           " " + getName() + " executable '" + getExecutable() + "' not found.");
52.     }
53.
54.
55.
56.
57.     public URI getExecutable() {
58.         return _executable;
59.     }
60.
61.
62.
63.
64.     public boolean executableExists() {
65.         final File file = new File(getExecutable().toString());
66.         return file.exists() && file.isFile();
67.     }
68.
69.
70.
71.
72.     public void scanFile(final File arg) throws
UploadRejectedException {
73.         if (Logging.isDebugEnabled(LOGGER)) {
74.             Logging.logDebug("scanning file " + arg, LOGGER); // NON-
NLS
75.         }
76.         final String[] args = {getExecutable().toString(),
arg.toString()};
77.         try {
78.             final StringWriter out = new StringWriter();
79.             final StringWriter error = new StringWriter();
80.             AccessUtil.executeProcess(out, error, args);
81.             final String message = out.toString();
82.             final String[] stat = message.split(":");
83.             final String[] cause = message.split("\n");
84.             if ( ! "OK".equals(stat[1].substring(0, 3)) ) {
85.                 Logging.logError("File upload rejected - " + message,
LOGGER); // NON-NLS
86.                 throw new UploadRejectedException(cause[0]);
```



```
87.         }
88.         if (Logging.isDebugEnabled(LOGGER)) {
89.             Logging.logDebug("scanned file " + arg, LOGGER);//
NON-NLS
90.         }
91.     } catch (IOException e) {
92.         Logging.logError("Couldn't start process", e, LOGGER);//
NON-NLS
93.         throw new UploadRejectedException("Couldn't start upload
filter process", e);
94.
95.     }
96. }
97.
98.
99.
100.
101.     public boolean isThreadSafe() {
102.         return true;
103.     }
104.
105. }
```

Listing 64: de.espirit.firstspirit.opt.vscan.engines.clamav.ClamScanEngine

3.17.8 Konfiguration und Persistenz – fs-vscan.conf

Die zentrale Konfigurationsdatei befindet sich im Archiv `fs-vscan.jar` und wird während des Modul-Installations-Prozesses in das FirstSpirit Modul-Konfigurationverzeichnis des VScan-Service (`conf/modules/FS VScan Service.VScanService`) ausgerollt (Verzeichnisstruktur siehe Kapitel 2.7 und 2.8 ab Seite 14).



3.18 Beispiel: Modul-Implementierung einer Komponente vom Typ WebApp

Dieses Kapitel soll die Entwicklung eines Moduls, bestehend aus einer einfachen WebApp-Komponente aufzeigen. Dabei werden ein einfaches Servlet und eine einfache Taglibrary entwickelt, die als Modul auf einem FirstSpirit-Server installiert werden können. Zusätzlich wird eine GUI implementiert, mit der Einstellungen für jede Umgebung (Vorschau/Staging/Produktion) vorgenommen werden können. Webapplikationen, die als Modul umgesetzt werden, können über FirstSpirit-Mechanismen auf verschiedenen Webservern installiert werden (siehe Dokumentation für Administratoren, Kapitel 7.3.16).

Der vollständige Quellcode ist im Zip-Archiv zum Modul-Entwicklerhandbuch (MODDEV4x_modexamples.zip) im Unterverzeichnis `examples/FS_V4_mod/webapp` enthalten (vgl. Kapitel 3.10 Seite 53).

3.18.1 Entwicklung der Webapplikation

Die Beispielimplementierung enthält ein einfaches Servlet und eine JSP-Taglibrary, die aus einem einzigen Tag besteht. Der gesamte Code ist auch ohne FirstSpirit lauffähig und kann auch durch eine entsprechende Konfiguration der IDE debugged bzw. getestet werden. Andere, ebenfalls auf Java basierende, Webtechnologien, wie z.B. JSP-Beans oder JavaServerFaces (JSF) können ebenfalls problemlos in ein Modul überführt werden.

Alle Java-Klassen der Webapplikation befinden sich unter `examples/webapp/src/web`.

3.18.1.1 Die Taglibrary

Die Taglibrary besteht nur aus einem einzigen „Hello-World“-Tag und enthält keinerlei FirstSpirit spezifischen Code. Es wurde das Interface `SimpleTagSupport` erweitert, welches Teil der JSP-Spezifikation 2.0 ist.

```
1. package de.espirit.firstspirit.opt.examples.webapp.web;
2.
3. import javax.servlet.jsp.JspException;
4. import javax.servlet.jsp.JspWriter;
5. import javax.servlet.jsp.PageContext;
6. import javax.servlet.jsp.tagext.SimpleTagSupport;
7.
```



```
8.    /**
9.     * Simple tag that returns a "Hello World" greeting
10.   *
11.   */
12.   public class HelloWorldTag extends SimpleTagSupport {
13.       private String name = "World";
14.
15.       public void setName(String name) {
16.           this.name = name;
17.       }
18.
19.       public void doTag() throws JspException {
20.           PageContext pageContext = (PageContext) getJspContext();
21.           JspWriter out = pageContext.getOut();
22.           try {
23.               out.println("Hello " + name);
24.           } catch (Exception e) {
25.               //Ignore
26.           }
27.       }
28.   }
```

Listing 65: Beispiel WebApp – Taglibrary

3.18.1.2 Der Tag-Library-Deskriptor

Um den JSP-Tag verwenden zu können, muss ein entsprechender Tag-Library-Deskriptor erstellt werden. Hier werden der Name des Tags und die implementierende Klasse definiert. Der Deskriptor befindet sich unter `examples/webapp/web/WEB-INF/HelloWorld.tld`.

```
1.    <?xml version="1.0" encoding="UTF-8"?>
2.    <taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
5.         jsptaglibrary_2_0.xsd">
6.
7.    <tlib-version>1.0</tlib-version>
8.    <short-name>helloworld</short-name>
9.    <uri>HelloWorld</uri>
10.
11.    <tag>
12.        <name>greeter</name>
13.        <tag-class>
```



```
    de.espirit.firstspirit.opt.examples.webapp.web.HelloWorldTag
    </tag-class>
10.    <body-content>empty</body-content>
11.    <attribute>
12.        <name>name</name>
13.        <rtexprvalue>>true</rtexprvalue>
14.        <required>>false</required>
15.    </attribute>
16. </tag>
17. </taglib>
```

Listing 66: Beispiel WebApp - Tag-Library-Deskriptor

3.18.1.3 Das Servlet

Das Servlet enthält ebenfalls keinerlei FirstSpirit spezifischen Code. Wird dem Servlet der GET-Parameter `name` übergeben, so wird der Benutzer mit diesem Namen begrüßt. Zusätzlich gibt das Servlet aus, wie oft es, seit dem letzten Start der Webapplikation, aufgerufen wurde. Um den Umgang mit einer Konfigurationsdatei zu demonstrieren, wird außerdem noch den Wert der Property `environment` ausgegeben. Der Wert wird über den Konfigurationsdialog des Moduls konfiguriert (siehe Kapitel 3.18.1.5 Seite 190).

```
1.    package de.espirit.firstspirit.opt.examples.webapp.web;
2.
3.    import javax.servlet.ServletException;
4.    import javax.servlet.http.HttpServlet;
5.    import javax.servlet.http.HttpServletRequest;
6.    import javax.servlet.http.HttpServletResponse;
7.    import java.io.IOException;
8.    import java.io.InputStream;
9.    import java.io.PrintWriter;
10.   import java.util.Properties;
11.
12.   /**
13.    * Simple HelloWorld Servlet
14.    *
15.    * Demonstrates the usage of:
16.    * - GET input param
17.    * - a property file
18.    */
19.   public class HelloWorldServlet extends HttpServlet {
20.       private int counter = 0;
```



```
21.     private String environment = "PRODUCTION";
22.     private Properties p;
23.
24.     public void init() {
25.         InputStream is = getServletContext().getResourceAsStream(
26.             "/WEB-INF/configuration.properties");
27.         Properties p = new Properties();
28.         try {
29.             p.load(is);
30.             environment = p.getProperty("environment");
31.             final String counterInitialValue =
32.                 p.getProperty("counterInitialValue");
33.             counter = Integer.parseInt(counterInitialValue);
34.         } catch (IOException e) {
35.             e.printStackTrace();
36.         }
37.     }
38.
39.     protected void doPost(HttpServletRequest request,
40.         HttpServletResponse response) throws ServletException,
41.         IOException {
42.         doGet(request, response);
43.     }
44.
45.     protected void doGet(javax.servlet.http.HttpServletRequest
46.         request, javax.servlet.http.HttpServletResponse response) throws
47.         javax.servlet.ServletException, IOException {
48.         PrintWriter writer = response.getWriter();
49.         writer.println("Environment is set to: " + environment);
50.
51.         if(null != request.getParameter("name")) {
52.             writer.println(new StringBuilder().append("Hello").append(
53.                 (request.getParameter("name")).append("!").toString());
54.         }
55.         else {
56.             writer.println("Hello Stranger!");
57.         }
58.         counter++;
59.         writer.println(new StringBuilder().append("This servlet has
60.             been called ").append(counter).append(" times.").toString());
61.     }
62. }
```



Listing 67: Beispiel WebApp – Servlet (Hello World)

3.18.1.4 Die web.xml

Zur Definition des Servlets (`HelloWorldServlet`) und des Servlet-Mappings wird noch eine `web.xml` benötigt. Die `web.xml` befindet sich unter `examples/webapp/web/WEB-INF/web.xml`.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >
3.
4.     <servlet>
5.         <servlet-name>HelloWorldServlet</servlet-name>
6.         <servlet-class>de.espirit.firstspirit.opt.examples.webapp.web.
           HelloWorldServlet</servlet-class>
7.     </servlet>
8.     <servlet-mapping>
9.         <servlet-name>HelloWorldServlet</servlet-name>
10.        <url-pattern>/hello</url-pattern>
11.    </servlet-mapping>
12. </web-app>
```

Listing 68: Beispiel WebApp – web.xml

Mit dieser Konfiguration kann das Servlet über folgende URL erreicht werden:

<http://YOURHOST/YOURAPP/hello>

Aufruf mit definierten `name` Parameter:

<http://YOURHOST/YOURAPP/hello?name=John>

3.18.1.5 Die Konfigurationsdatei

Die Konfigurationsdatei enthält zwei Properties, die vom Servlet ausgelesen werden. Das Servlet erwartet die Konfigurationsdatei innerhalb des `WEB-INF`-Verzeichnisses der Webapplikation. Die Datei befindet sich also unter `examples/webapp/web/WEB-INF/configuration.properties`.

```
environment=PRODUCTION
counterInitialValue=0
```



Listing 69: Beispiel WebApp – Konfigurationsdatei

Die Property `environment` kann verwendet werden, um das Loglevel für unterschiedliche Umgebungen (Entwicklungsumgebung, Testumgebung, Produktionsumgebung) zu setzen. In der momentanen Implementierung wird lediglich der Wert ausgegeben.

Die Property `counterInitialValue` definiert den Startwert des Zugriffszählers. Der Zugriffszähler ist nur als statische Variable umgesetzt, deren Wert nicht persistiert wird. Ein erneuter Start der Webapplikation setzt den Zähler also wieder auf den hier definierten Wert zurück.

3.18.2 FirstSpirit spezifische Klassen

Die folgenden Klassen sind für die Initialisierung und Konfiguration des Moduls bzw. der Komponente auf dem FirstSpirit-Server notwendig. Alle Klassen befinden sich unter `examples/webapp/impl`.

3.18.2.1 WebApp.java

WebApp-Komponenten werden beim Start des FirstSpirit-Servers geladen. Hierfür muss eine Klasse implementiert werden, die von `AbstractWebApp` abgeleitet wird (Details zur Initialisierung von Komponenten siehe Kapitel 3.4 Seite 36).

Die Beispielimplementierung enthält keinerlei Logik und soll nur die vorhandenen Methoden aufzeigen. Durch die Implementierung von `installed`, `updated`, `uninstalled` kann der Modul-Entwickler auf die entsprechenden Ereignisse reagieren. Typische Anwendungsfälle sind die Installation bzw. Deinstallation von Knoten (Vorlagen/Medien etc.) im zugehörigen Projekt oder die automatische Anpassung von Konfigurationen während der Aktualisierung des Moduls.

```
1. package de.espirit.firstspirit.opt.examples.webapp.configuration;
2.
3. import de.espirit.firstspirit.module.AbstractWebApp;
4.
5. /**
6.  * We need to implement the web application interface for your webapp
7.  * component. More advanced webapplication will use the methods to
8.  * install/update/delete additional elements.
9.  */
10. public class WebApp extends AbstractWebApp {
```



```
10.  /**
11.   * This method will be called if the component has been installed
12.   * successfully.
13.   * Use it to install additional elements
14.   */
15.   @Override
16.   public void installed() {
17.       super.installed();
18.   }
19.
20.  /**
21.   * Called when component was uninstalled.
22.   * Use this method to cleanup any files that might have been
23.   * created by your component.
24.   */
25.   @Override
26.   public void uninstalling() {
27.       super.uninstalling();
28.   }
29.
30.  /**
31.   * Called when component has been updated successfully
32.   */
33.   @Override
34.   public void updated(String oldVersionString) {
35.       super.updated(oldVersionString);
36.   }
37.
38.   @Override
39.   public void createWar() {
40.       super.createWar();
41.   }
```

Listing 70: Beispiel WebApp – Implementierung WebApp

3.18.2.2 WebAppConfiguration.java

In dieser Klasse wird die Konfigurationsoberfläche für das Modul definiert. Es besteht aus einem einfachen Swing-Layout und enthält zwei Textfelder, über welche die Werte der `configuration.properties` Datei verändert werden können. Die Implementierung einer Konfigurationsklasse ist optional, kann also entfallen, sofern eine WebApp-Komponente keine Konfigurationsmöglichkeiten zur Verfügung stellen



soll.

Die main-Methode dient lediglich der einfacheren GUI-Entwicklung, muss also nicht zwingend implementiert werden.

```
1. package de.espirit.firstspirit.opt.examples.webapp.configuration;
2.
3. import de.espirit.common.Logging;
4. import de.espirit.firstspirit.io.FileHandle;
5. import de.espirit.firstspirit.module.Configuration;
6. import de.espirit.firstspirit.module.WebEnvironment;
7. import net.java.dev.designgridlayout.DesignGridLayout;
8.
9. import javax.swing.*;
10. import java.awt.*;
11. import java.awt.event.ActionEvent;
12. import java.awt.event.ActionListener;
13. import java.beans.PropertyChangeListener;
14. import java.io.ByteArrayInputStream;
15. import java.io.ByteArrayOutputStream;
16. import java.io.IOException;
17. import java.util.*;
18. import java.util.List;
19.
20. /**
21.  * Implements the configuration dialog
22.  */
23. public class WebAppConfiguration implements
24. Configuration<WebEnvironment>{
25.     /**
26.      * name of the configuration file
27.      */
28.     public static final String CONFIGURATION_FILE =
29.         "configuration.properties";
30.     /**
31.      * name of the configuration property
32.      */
33.     public static final String PROP_ENVIRONMENT = "environment";
34.     public static final String PROP_COUNTER_INITIAL_VALUE =
35.         "counterInitialValue";
36.
37.     private final Properties _config;
38.     private WebEnvironment _env;
```



```
36.     private JComponent _gui;
37.
38.     private JTextField _environment;
39.     private JTextField _counterInitialValue;
40.
41.
42.     public WebAppConfiguration() {
43.         _config = new Properties();
44.     }
45.
46.     public boolean hasGui() {
47.         return true; // We want to have a GUI
48.     }
49.
50.     public JComponent getGui(final Frame frame) {
51.         if(_gui == null) {
52.             final JPanel mainPanel = new JPanel();
53.             final DesignGridLayout layout = new
54.                 DesignGridLayout(mainPanel);
55.             mainPanel.setOpaque(false);
56.
57.             final JLabel headline1 = new JLabel("WebApp
58.                 Configuration");
59.             final Font font = headline1.getFont();
60.             headline1.setFont(font.deriveFont(font.getStyle() ^
61.                 Font.BOLD));
62.             layout.row().left().fill().add(headline1);
63.
64.             final JLabel lEnvironment = new JLabel("Environment");
65.             _environment = new JTextField(_config.getProperty(
66.                 PROP_ENVIRONMENT));
67.             layout.row().grid(lEnvironment).add(_environment, 4);
68.
69.             final JLabel lInitialCount = new JLabel("Counter Initial
70.                 Value");
71.             _counterInitialValue = new TextField(_config.getProperty(
72.                 PROP_COUNTER_INITIAL_VALUE));
73.             layout.row().grid(lInitialCount).add(_counterInitialValue,
74.                 4);
75.
76.             _gui = mainPanel;
77.         }
78.         return _gui;
79.     }
80. }
```



```
72.     }
73.
74.     /**
75.      * load configuration from configuration file
76.      */
77.     public void load() {
78.
79.         //set default value to PRODUCTION
80.         _config.setProperty(PROP_ENVIRONMENT, "PRODUCTION");
81.         _config.setProperty(PROP_COUNTER_INITIAL_VALUE, "0");
82.         try {
83.             final FileHandle iniFile = getEnvironment().getConfDir().
84.                 obtain(CONFIGURATION_FILE);
85.             if(iniFile.isFile()) {
86.                 _config.load(iniFile.load());
87.             }
88.         } catch (IOException e) {
89.             Logging.logWarning("Couldn't load configuration file!", e,
90.                 WebAppConfiguration.class);
91.         }
92.         if(_gui != null) {
93.             _environment.setText(_config.getProperty(PROP_ENVIRONMENT));
94.             _counterInitialValue.setText(_config.getProperty(
95.                 PROP_COUNTER_INITIAL_VALUE));
96.         }
97.     }
98.
99.     /**
100.      * read values from gui and store them into the configuration file
101.      */
102.     public void store() {
103.
104.         if(_gui != null) {
105.             _config.setProperty(PROP_ENVIRONMENT,
106.                 _environment.getText());
107.             _config.setProperty(PROP_COUNTER_INITIAL_VALUE,
108.                 _counterInitialValue.getText());
109.         }
110.
111.         try {
112.             final ByteArrayOutputStream out = new
113.                 ByteArrayOutputStream();
114.             _config.store(out, "FirstSpirit WebApp Example Module -
```



```
        Configuration");
109.     final FileHandle iniFile =getEnvironment().getConfDir().
        obtain(CONFIGURATION_FILE);
110.     iniFile.save(new ByteArrayInputStream(out.toByteArray()));
111.     } catch (IOException e) {
112.         Logging.logWarning("Couldn't store configuration file!",
        e, WebAppConfiguration.class);
113.     }
114. }
115.
116. public Set<String> getParameterNames() {
117.     final Set<String> result = new HashSet<String>();
118.     for (final Enumeration<?> names =
        _config.propertyNames(); names.hasMoreElements();) {
119.         result.add(names.nextElement().toString());
120.     }
121.     return result;
122. }
123.
124. public String getParameter(final String name) {
125.     return _config.getProperty(name);
126. }
127.
128. public void init(final String moduleName, final String
    componentName, final WebEnvironment env) {
129.     _env = env;
130. }
131.
132. public WebEnvironment getEnvironment() {
133.     return _env;
134. }
135.
136. /**
137.  * Main method for easier gui development.
138.  *
139.  * @paramargs unused.
140.  */
141. public static void main(final String[] args) {
142.     final JFrame frame = new JFrame("Configuration");
143.
144.     final WebAppConfiguration configurator = new
        WebAppConfiguration();
145.     final JComponent gui = configurator.getGui(frame);
```



```
146.  
147.     final JButton closeBtn = new JButton("Close");  
148.     closeBtn.addActionListener(new ActionListener() {  
149.  
150.         public void actionPerformed(final ActionEvent e) {  
151.             frame.setVisible(false);  
152.             frame.dispose();  
153.         }  
154.     });  
155.  
156.     frame.getContentPane().add(gui);  
157.     frame.getRootPane().setDefaultButton(closeBtn);  
158.     frame.setSize(580, 300);  
159.     frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);  
160.     frame.setVisible(true);  
161. }  
162.  
163. }
```

Listing 71: Beispiel WebApp – Implementierung WebAppConfiguration

3.18.2.3 WebAppModule.java

Mit dieser Klasse kann auf bestimmte Ereignisse, wie Installation, Deinstallation oder Aktualisierung des Modules durch den Benutzer in der Administrationsoberfläche, reagiert werden. Über die Methoden können Änderungen an allen Komponenten des Moduls vorgenommen werden. Da die Beispielimplementierung nur eine Komponente enthält, wurde hier keine Logik implementiert. Wird diese Funktionalität nicht benötigt, kann die Klasse (und der Eintrag in der module.xml) vollständig entfallen.

```
1.     package de.espirit.firstspirit.opt.examples.webapp.configuration;  
2.  
3.     import de.espirit.firstspirit.module.Module;  
4.     import de.espirit.firstspirit.module.ServerEnvironment;  
5.     import de.espirit.firstspirit.module.descriptor.ModuleDescriptor;  
6.  
7.     /**  
8.      * You don't need this class if you just want to create a simple  
9.      * webapp, but it might be useful if your module will contain multiple  
10.     * components.  
11.     */  
12.     public class WebAppModule implements Module {
```



```
13.     /*
14.      *   Initializes this component with the given descriptor and
15.      *   environment.
16.      */
17.     public void init(final ModuleDescriptor descriptor, final
18.         ServerEnvironment env) {
19.     }
20.     /*
21.      *   Use this method to install templates or create other elements
22.      */
23.     public void installed() {
24.     }
25.     /*
26.      *   Use this method to uninstall templates and to perform a cleanup
27.      */
28.     public void uninstalling() {
29.     }
30.     /*
31.      *   Use this method to update templates or alter configuration
32.      *   settings during an update
33.      */
34.     public void updated(final String oldVersionString) {
35.     }
36.     @Override
37.     public String toString() {
38.         return "FirstSpirit WebApp Example Module";
39.     }
40.     }
41. }
```

Listing 72: Beispiel WebApp – Implementierung WebAppModule

3.18.2.4 Der Modul-Deskriptor

Der Modul-Deskriptor (module.xml) beschreibt das Modul und die darin enthaltenen Komponenten, sowie deren Ressourcen (siehe Kapitel 3.8 Seite 39 und Kapitel 3.9.1.8 Seite 51).



Die module.xml befindet sich unter `examples/webapp/module.xml`.

```
1.  <!DOCTYPE module SYSTEM "../lib/module.dtd">
2.  <module>
3.      <name>FirstSpirit WebApp Example Module</name>
4.      <version>@VERSION@</version>
5.      <description>FirstSpirit WebApp Example Module</description>
6.      <vendor>e-Spirit AG</vendor>
7.      <class>
8.      de.espirit.firstspirit.opt.examples.webapp.configuration.WebAppModule
9.      </class>
10.     <components>
11.         <web-app>
12.             <name>FirstSpirit WebApp Example Module</name>
13.             <description>Web component of FIRSTspirit
14.                 integration.</description>
15.             <class>
16.             de.espirit.firstspirit.opt.examples.webapp.configuration.WebApp
17.             </class>
18.             <configurable>
19.             de.espirit.firstspirit.opt.examples.webapp.configuration.WebAppConfiguratio
20.             n
21.             </configurable>
22.             <web-xml>web.xml</web-xml>
23.             <resources>
24.                 <resource>lib/webapp-example-
25.                     @VERSION@.jar</resource>
26.             </resources>
27.             <web-resources>
28.                 <resource>HelloWorld.tld</resource>
29.                 <resource>configuration.properties</resource>
30.                 <resource>
31.                     lib/webapp-example-@VERSION@-webapp.jar
32.                 </resource>
33.             </web-resources>
34.         </web-app>
35.     </components>
36. </module>
```

Listing 73: Beispiel WebApp – Modul-Deskriptor



3.18.3 Anmerkungen zum Ant-Build-File

Im Rahmen der Beispielimplementierung wurden Ant-Tasks erstellt, die das notwendige FSM-Archiv erstellen.

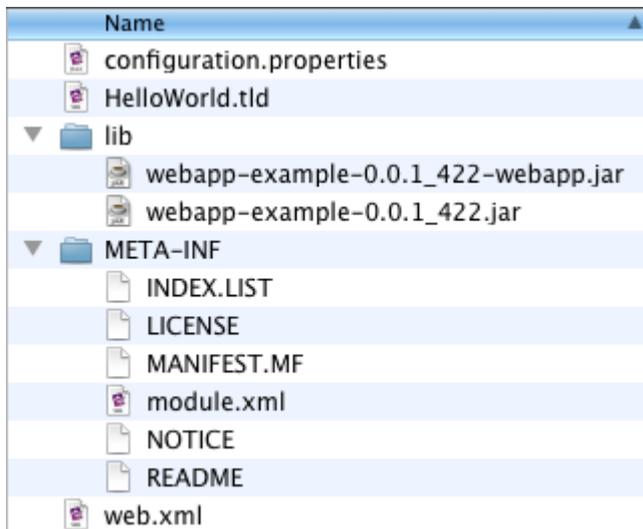


Abbildung 3-16: Beispiel WebApp – Verzeichnisstruktur FSM

Das FSM kann mittels `ant assemble-fsm` erzeugt werden.

3.18.4 Installation des Modules

Das erzeugte FSM-Archiv kann, wie jedes andere Modul, über die FirstSpirit Server- und Projektkonfiguration installiert und konfiguriert werden (siehe FirstSpirit Handbuch für Administratoren, Kapitel 7.3.14 Module und Kapitel 7.3.16 Web-Applikationen).



3.19 Namensgleichheit bei Modul-Ressourcen (z.B. Zip-Exportdateien)

Existieren verschiedene Module mit namensgleichen Ressourcen, z.B. `myZipExport.zip` in Modul01 und Modul02 z.B. jeweils in den Modulverzeichnissen `files` innerhalb eines FSM-Archives, wird durch den Server-Classloader die erste gefundene Ressource mit dem Dateinamen `myZipExport.zip` geladen, also analog zur Java-VM. Wenn mehrere gleichnamige Jars im Classpath liegen, gewinnt die erste gefundene Ressource. Grundlegend sind Modul-Ressourcen systemweit verfügbar. D.h., sie werden in ein temporäres Verzeichnis (eines pro Modul) entpackt und sind dann über den Server-ClassLoader erreichbar. Das führt dazu, dass bei Namensgleichheit von Dateien das erste Modul gewinnt bzw. dessen Ressource `myZipExport.zip` „gewinnt“.

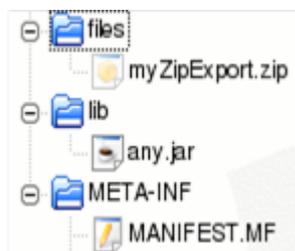


Abbildung 3-17: Namensgleiche Modul-Ressourcen

Um diese Limitierung zu umgehen, ist der empfohlene Weg, die Dateien mit ins Jar neben die Klasse zu legen, die diese Dateien (`myZipExport.zip`) benötigt. Mit `MyClass.class.getResourceAsStream(...)` können diese Ressourcen anschließend konfliktlos geladen werden, in diesem Fall also `ZipImporter.class.getResourceAsStream(myZipExport.zip);`.

Nicht zu verwechseln mit `getResource(...)`; hier wird eine URL zurückgegeben, welche Remote auf dem FirstSpirit-Server nicht zum temporären Verzeichnis umgesetzt wird.



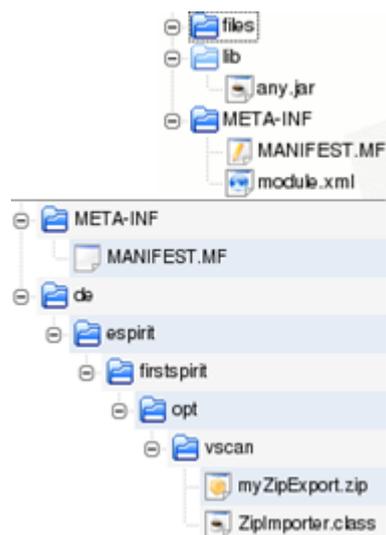


Abbildung 3-18: Laden namensgleicher Modul-Ressourcen



3.20 Erzeugung von FSM-Archiven (.fsm)

```

1.   <target name="main-fsm" depends="main-jar " description="Builds the
    main module file (*.fsm).">
2.       <mkdir dir="/tmp/myModuleName/lib"/>
3.       <mkdir dir="/tmp/myModuleName/META-INF"/>
4.       <copy file="Path to the previous created Jar
    File/myModuleName.jar" todir="/tmp/myModuleName/lib"
    preservelastmodified="true" overwrite="true" />
5.       <copy file="module.xml" tofile="/tmp/myModuleName/META-
    INF/module.xml" overwrite="true">
6.           <filterset>
7.               <filter token="VERSION" value="1.1.1"/>
8.           </filterset>
9.       </copy>
10.      <jar jarfile="Output dir path/myModuleName.fsm"
    basedir="/tmp/myModuleName" excludes="**/*.dependency" duplicate="preserve"
    keepcompression="true"/>
11.  </target>

```

Listing 74: FSM-Archiv-Erzeugung – build.xml

Ein komplettes Beispiel einer build.xml Datei findet sich in Kapitel 3.14.11 (Seite 147).

3.21 Signieren von Modulen – Jar-Archiv-Klassen

Signierte Module werden bei der Installation überprüft und gegebenenfalls zurückgewiesen. Die Signierungs-Operation lässt sich beispielsweise mit folgendem Ant-Aufruf in die build.xml (Kapitel 3.14.11 Seite 147) integrieren.

```

1.   <property name="signkey" value="yourKeySigningCompany"/>
2.   <signjar jar="themodule.jar" alias="${signkey}" storepass="yourPass"
    keystore="yourKeyFile.jks"/>

```

Listing 75: Signieren von Modulen mit einer Ant-Task

alias	Auswahl des Private-Keys (Zertifikat) anhand des eindeutigen Bezeichners innerhalb der Keystore-Datei.
storepass	Passwort für die Keystore-Integrität (Private-Key Passwort)
keystore	Key-Archive („Schlüsselbund“), Keystore-Datei, die ggf. mehrere Zertifikate enthält.



3.22 Internationalisierung von Modulen – i18n

Generell ist die aktuelle selektierte Sprache des FirstSpirit JavaClients oder der Server- und Projektkonfiguration über `Locale.getDefault()` verfügbar.



Eine Beispiel-Implementierung könnte wie folgt aussehen:

```
1. package de.espirit.firstspirit.opt.vscan.resources;
2.
3. import de.espirit.common.base.Logging;
4.
5. import javax.swing.Icon;
6. import javax.swing.ImageIcon;
7. import java.awt.Image;
8. import java.awt.Toolkit;
9. import java.io.IOException;
10. import java.io.InputStream;
11. import java.util.Locale;
12. import java.util.MissingResourceException;
13. import java.util.ResourceBundle;
14.
15.
16. /**
17.  * $Date: 2008-05-26 15:08:34 +0200 (Mo, 26 Mai 2008) $
18.  *
19.  * @version $Revision: 22537 $
20.  */
21. public final class ModuleResources {
22.
23.     public static final Class LOGGER = ModuleResources.class;
24.
25.     private static volatile ResourceBundle _bundle;
26.     private static final String LOCALE_RESOURCES_PKG =
27.         "de.espirit.firstspirit.opt.vscan.resources.locale.Messages"; // NON-NLS
28.     private static final String ICON_RESOURCES_PKG =
29.         "/de/espirit/firstspirit/opt/vscan/resources/icons"; // NON-NLS
30.     private static final byte[] BYTE = new byte[0];
31.
32.
33.     private ModuleResources() {
34.     }
35.
36.
37.
38.
39.     public static ResourceBundle getResourceBundle() {
```



```
40.         Logging.logInfo("Loading locale properties for '" +
Locale.getDefault() + "'", LOGGER); // NON-NLS
41.
42.         if (_bundle != null) {
43.             return _bundle;
44.         }
45.
46.         //noinspection UnusedCatchParameter
47.         try {
48.             _bundle = ResourceBundle.getBundle(LOCALE_RESOURCES_PKG,
Locale.getDefault());
49.         } catch (MissingResourceException e) {
50.             throw new MissingResourceException("Missing Resource
bundle: " + Locale.getDefault() + " ", LOCALE_RESOURCES_PKG, "");
51.         }
52.
53.         return _bundle;
54.     }
55.
56.
57.
58.
59.     public static ResourceBundle getResourceBundle(final String
localeResourceFolder) {
60.         Logging.logInfo("Loading locale properties for '" +
Locale.getDefault() + "'", LOGGER); // NON-NLS
61.         //noinspection UnusedCatchParameter
62.         try {
63.             _bundle = ResourceBundle.getBundle(localeResourceFolder,
Locale.getDefault());
64.         } catch (MissingResourceException e) {
65.             throw new MissingResourceException("Missing Resource
bundle: " + Locale.getDefault() + " ", localeResourceFolder, "");
66.         }
67.
68.         return _bundle;
69.     }
70.
71.
72.
73.
74.     @SuppressWarnings({"UnusedCatchParameter"})
75.     public static String getString(final String key) {
```



```
76.         try {
77.             if (_bundle == null) {
78.                 getResourceBundle();
79.             }
80.             return _bundle.getString(key);
81.         } catch (MissingResourceException e) {
82.             throw new MissingResourceException("Missing Resource: " +
Locale.getDefault() + " - key: " + key + " - resources: " +
LOCALE_RESOURCES_PKG, LOCALE_RESOURCES_PKG, key);
83.         } catch (NullPointerException e) {
84.             throw new NullPointerException("No bundle set: use
ModuleResources.getResourceBundle().getString(...)");
85.         }
86.
87.     }
88.
89.
90.
91.
92.     private static InputStream getResourceStream(final String
iconResourcePkg, final String filename) {
93.         String iconResourcePkg1 = iconResourcePkg;
94.         if (iconResourcePkg1.startsWith("/")) {
95.             iconResourcePkg1 = iconResourcePkg1.replaceFirst("/", "");
96.         }
97.
98.         final String resname = "/" + iconResourcePkg1.replace('.',
'/') + "/" + filename;
99.         final Class clazz = ModuleResources.class;
100.
101.         return clazz.getResourceAsStream(resname);
102.     }
103.
104.
105.
106.
107. }
```

Listing 76: Internationalisierung ModuleResources.java





(siehe Datei "ModulResources.java", enthalten in der ZIP-Datei mit der Beispiel-Implementierung, FirstSpirit Online-Dokumentation, Kapitel „Dokumentation“ / „Für Entwickler“).

Ein exemplarisches Anwendungsbeispiel für die Nutzung findet sich in der Konfigurationsoberflächen-Klasse (VScanServiceConfigPanel, siehe Seite 172, Klassen-Beschreibung VScanServiceConfigPanel).

```
ModuleResources.getString("fs-  
resource.module.vscan.admin.gui.VScanServiceConfigPanel_EngineSetu  
p")
```

Hierbei werden immer die Ressourcen aus dem Package `de.espirit.firstspirit.opt.vscan.resources.locale` verwendet. Um andere Ressourcen zu nutzen, muss der Pfad hier über die Methode explizit angegeben werden. Hier steht die Methode

```
public static ResourceBundle getResourceBundle(final String  
localeResourcePath)
```

zur Verfügung, über die sich ein Package-„Pfad“ übergeben lässt. Ein Aufruf würde sich beispielsweise wie folgt darstellen:

```
ModuleResources.getResourceBundle(other.resources.locale).getStrin  
g("fs-  
resource.module.vscan.admin.gui.VScanServiceConfigPanel_EngineSetu  
p")
```



3.23 Icon Ressourcen

Für die Ablage von Bildern ist das Package

`de.espirit.firstspirit.opt.vscan.resources` vorgesehen.

```
ModuleResources.loadIcon("myImage.png")
```

Laden der Ressourcen aus dem default-Package für Icons

(`de.espirit.firstspirit.opt.vscan.resources.icons`).

```
ModuleResources.loadIcon("de.espirit.firstspirit.opt.vscan.resources.icons", "myImage.png")
```

Laden der Ressourcen aus einer beliebigen Quelle im JAR order außerhalb des Archivs.

3.24 Text Ressourcen

Mithilfe der Methode `loadTextResource` können Text-Dateien geladen werden:

```
public static String loadTextResource(final String pkgName, final String filename)
```

3.25 Integration von Eingabekomponenten (Editoren) in den JavaClient

3.25.1 Beispiel Formular-Element (GOM-Form)

Editoren nutzen für die Integration der Eingabekomponente das FirstSpirit GUI Object Model (GOM). Hierbei wird über die Definition eines XML-Identifiers, z.B. `CUSTOM_TEXTAREA` (siehe auch Kapitel 3.11 Seite 37), im Formular-Element (GOM-Form) (`de.espirit.firstspirit.access.store.templatestore.gom.GomFormElement`) die Komponente im FirstSpirit JavaClient eingebunden. Eingabekomponenten in FirstSpirit dienen dazu, das Einpflegen und Bearbeiten von Inhalten für Redakteure so komfortabel wie möglich zu gestalten. Die gewünschten Eingabekomponenten werden vom Vorlagenentwickler im Registerbereich „Formular“ der Vorlage eingebunden (siehe auch Kapitel 3.11 Seite 56).

Für die einfache Editoren-Komponente (Kapitel 3.14 Seite 126) stellt sich die GOM-Form (`gui.xml`) wie folgt dar:

```
1. <CMS_MODULE>
```



```

2.     <CUSTOM_TEXTAREA name="myEditor" maxRows="10" hFill="yes" >
3.         <LANGINFOS>
4.             <LANGINFO lang="*" label="FirstSpirit Editor Example"/>
5.             <LANGINFO lang="DE" label="DE:FirstSpirit Editor Beispiel"/>
6.             <LANGINFO lang="EN" label="EN:FirstSpirit Editor Example"/>
7.         </LANGINFOS>
8.     </CUSTOM_TEXTAREA>
9. </CMS_MODULE>
    
```

Listing 77: Beispiel GOM-Form

Der Parameter `myEditor` ist hier der eindeutige Bezeichner der Komponente innerhalb des GOM und der Ausgabe-Kanäle (beispielsweise HTML, PDF). Weitere Parameter wie `hFill` sind optional, sie finden sich in den Hauptabschnitten der Online Dokumentation für FirstSpirit V4.0¹⁰ / Vorlagenentwicklung / Formulare.



Form-Elemente müssen ein Fallback-Label definieren <LANGINFO lang=""*

3.26 Modulansicht in der Server -und Projektkonfiguration

Nach erfolgter Installation des zuvor erstellen FSM-Archives stellt sich das installierte Modul in der FirstSpirit Server -und Projektkonfiguration wie folgt dar:

Modules			
Name	Version	Type	Scope
FirstSpirit Simple Editor Example	4.0_DEV.4069_19...		
└─ CMS_INPUT_SIMPLE_EDITOR_CONTENT	4.0_DEV.4069_19...	Editor	Global

Abbildung 3-19: Modulansicht des installierten Simple Editor Example Moduls in der Server -und Projektkonfiguration



Abbildung 3-20: Mögliche, für das Simple Editor Example Modul ausführbare Aktionen

¹⁰ [4] Online Dokumentation für FirstSpirit – ODFS



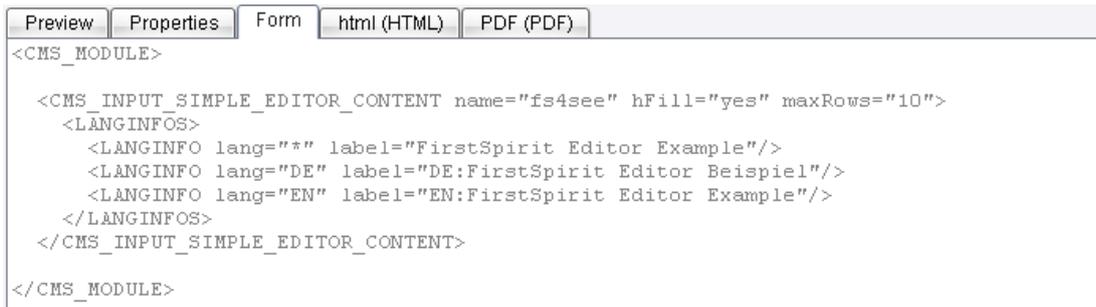


Abbildung 3-21: Mögliche, für das Simple Editor Example Modul ausführbare Aktionen in der FirstSpirit Version 4.1



3.27 Ansicht Komponente Formular und Ausgabe-Kanäle (bspw. HTML, PDF)

Um die Editor-Komponenten im FirstSpirit JavaClient zur Verfügung zu stellen, wird in der Vorlagen-Verwaltung (JavaClient) das Formular definiert (siehe Kapitel 3.25.1 Seite 209).



```

<CMS_MODULE>
  <CMS_INPUT_SIMPLE_EDITOR_CONTENT name="fs4see" hFill="yes" maxRows="10">
    <LANGINFOS>
      <LANGINFO lang="" label="FirstSpirit Editor Example"/>
      <LANGINFO lang="DE" label="DE:FirstSpirit Editor Beispiel"/>
      <LANGINFO lang="EN" label="EN:FirstSpirit Editor Example"/>
    </LANGINFOS>
  </CMS_INPUT_SIMPLE_EDITOR_CONTENT>
</CMS_MODULE>

```

Abbildung 3-22: Definition des Formulars für das Simple Editor Example Modul



```

<CMS_HEADER>
</CMS_HEADER>
<div>FirstSpirit4 Editor Example</div>
<p>${CMS_VALUE(fs4see)}</p>

```

Abbildung 3-23: Definition der HTML-Darstellung (Ausgabekanal) für das Simple Editor Example Modul



```

<CMS_HEADER>
</CMS_HEADER>
${CMS_IF(!fs4see.isEmpty)}$
  <fo:block font-size="10pt" color="#3864B4" font-weight="bold" space-after="1mm">
    ${CMS_VALUE(fs4see)}$
  </fo:block>
${CMS_END_IF}$

```

Abbildung 3-24: Definition des PDF-Ausgabekanals mittels Formatting Objects Processor innerhalb der FirstSpirit-Template-Syntax



4 Listings

LISTING 1: RESSOURCEN IM MODULE-, KOMPONENTEN-DESCRIPTOR.....	11
LISTING 2: VERSIONIERUNG VON RESSOURCEN IM MODUL-, KOMPONENTEN-DESCRIPTOR	12
LISTING 3: GLOBALES LOGGING	32
LISTING 4: LOGGING AUF MODULEBENE	32
LISTING 5: SERVERENVIRONMENT – SERVICE.....	33
LISTING 6, ZU FINDEN IN DER FIRSTSPIRIT-ACCESS-API	34
LISTING 7: KOMPONENTEN EVENT-METHODEN – INSTALLED, UPDATED, UNINSTALLED.....	38
LISTING 8: MODUL-DESKRIPTOR-BEISPIEL	39
LISTING 9: PFLICHTFELDER DES MODUL-DESKRIPTORS	40
LISTING 10: OPTIONALE MODUL-DESKRIPTOR-ELEMENTE	41
LISTING 11: LIBRARY KOMPONENTEN-DESKRIPTOR UND EIGENSCHAFTEN	42
LISTING 12: EDITOR KOMPONENTEN-DESKRIPTOR UND EIGENSCHAFTEN	43
LISTING 13: SERVICE KOMPONENTEN-DESKRIPTOR UND EIGENSCHAFTEN.....	45
LISTING 14: PUBLIC KOMPONENTEN-DESKRIPTOR UND EIGENSCHAFTEN	46
LISTING 15: ANPASSEN DER UMGEBUNGSVARIABLEN FÜR DAS ECLIPSE MODUL PROJECT BEISPIEL.....	55
LISTING 16: GOM-TYPISIERUNG UND MAPPING.....	59
LISTING 17: GOM – DIREKTE VERWENDUNG VON KINDELEMENTEN	60
LISTING 18: GOM MENGENWERTIGE VERWENDUNG (ELEMENTLISTEN).....	62
LISTING 19: GOM BEISPIEL COMBOBOX	66
LISTING 20: GOM-FORM REPRESENTATION	67
LISTING 21: REFERENZEN - ERZEUGEN EINER NEUEN REFERENZ (ÜBER REFERENCEHOLDER).....	114
LISTING 22: REFERENZEN - CHANGECALLBACK BEI DER ÄNDERUNG EINER REFERENZ.....	114
LISTING 23: REFERENZEN - VALUEENGINEERCONTEXT ÜBERGEBEN (VALUEENGINEERFACTORY)	115
LISTING 24: REFERENZEN – SPECIALISTSBROKER ÜBER VALUEENGINEERCONTEXT ANFORDERN	115
LISTING 25: REFERENZEN - REFERENCECONSTRUCTIONAGENT ANFORDERN	116
LISTING 26: REFERENZEN - REFERENCEHOLDER ERZEUGEN (VALUEENGINEER-IMPL.)	117
LISTING 27: REFERENZEN - REFERENCETRANSFORMATIONAGENT ANFORDERN	117
LISTING 28: REFERENZEN - REFERENCETRANSFORMATIONAGENT – READ(...).....	118
LISTING 29: REFERENZEN - REFERENCETRANSFORMATIONAGENT – READ(...).....	118
LISTING 30: REFERENZEN - REFERENCETRANSFORMATIONAGENT - WRITE(...)	119
LISTING 31: BEISPIEL GOMFORM – ERWEITERN MIT DER ABSTRAKTEN BASISIMPLEMENTIERUNG	127
LISTING 32: BEISPIEL GOMFORM – DEFINITION EINES XML-IDENTIFIERS	127
LISTING 33: BEISPIEL GOMFORM – DEFINITION VON ATTRIBUTEN	129
LISTING 34: BEISPIEL - XML-REPRÄSENTATION DER EINGABEKOMPONENTE IM JAVACLIENT	129
LISTING 35: BEISPIEL SWINGGADGETFACTORY – ERZEUGEN EINES TYP. SWINGGADGETS	130
LISTING 36: BEISPIEL SWINGGADGET – EINBINDEN DES BASISSTYPS SWINGGADGET	130
LISTING 37: BEISPIEL SWINGGADGET – EINBINDEN DER ASPEKTE VALUEHOLDER UND EDITABLE	130
LISTING 38: BEISPIEL SWINGGADGET – ERWEITERUNG MIT DER ABSTR BASISIMPLEMENTIERUNG	130
LISTING 39: BEISPIEL SWINGGADGET – KONSTRUKTOR.....	131
LISTING 40: BEISPIEL SWINGGADGET – IMPLEMENTIERUNG DER TEXTEINGABEKOMPONENTE.....	131
LISTING 41: BEISPIEL SWINGGADGET - IMPLEMENTIERUNG DES ASPEKTS VALUEHOLDER	132



LISTING 42: BEISPIEL SWINGGADGET - IMPLEMENTIERUNG DES ASPEKTS EDITABLE132

LISTING 43: BEISPIEL ÄNDERUNGEN PROPAGIEREN - HINZUFÜGEN EINES DOCUMENTLISTENERS..133

LISTING 44: BEISPIEL ÄNDERUNGEN PROPAGIEREN – AUFRUF VON NOTIFYVALUECHANGE()133

LISTING 45: BEISPIEL ÄNDERUNGEN PROPAGIEREN – DER ASPEKT VALUELIKENING136

LISTING 46: BEISPIEL VALUEENGINEERFACTORY – LESEN U. SCHREIBEN DES PERSISTENZTYPUS ..137

LISTING 47: BEISPIEL VALUEENGINEER(IMPLEMENTIERUNG)137

LISTING 48: BEISPIEL VALUEENGINEER-IMPL. – METHODE READ(...) – XML TO OBJECT.....138

LISTING 49: BEISPIEL VALUEENGINEER-IMPL. – METHODE WRITE(...) –OBJECT TO XML139

LISTING 50: BEISPIEL VALUEENGINEER-IMPL. – LEERWERT-BEHANDLUNG139

LISTING 51: BEISPIEL VALUEENGINEER-IMPL. – KOPIEREN DES PERSISTENZTYPUS140

LISTING 52: BEISPIEL VALUEENGINEER-IMPL. – ERZEUGEN EINER ASPEKTORIENTIERTEN INSTANZ141

LISTING 53: BEISPIEL VALUEENGINEER-IMPL. – VALUE-ASPEKT MATCHSUPPORTING142

LISTING 54: BEISPIEL – CONTENTHIGHLIGHTING AKTIVIEREN.....142

LISTING 55: BEISPIELSWINGGADGET – DER ASPEKT HIGHLIGHTABLE145

LISTING 56: BEISPIEL - MODUL- UND KOMPONENTEN-DESKRIPTOR.....146

LISTING 57: ANT BUILD.XML MIT DEN TARGETS MAIN-JAR, MAIN-FSM, DEPLOY148

LISTING 58: MODUL-DESKRIPTOR OHNE KOMPONENTEN – MODULE.XML150

LISTING 59: MODUL-DESKRIPTOR MIT LIBRARY-KOMPONENTE – MODULE.XML153

LISTING 60: DREI KOMPONENTEN-TYPEN MODUL-DESKRIPTOR159

LISTING 61: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANSERVICEIMPL170

LISTING 62: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANSERVICECONFIGPANEL179

LISTING 63: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANFILTERPROXY182

LISTING 64: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.ENGINES.CLAMAV.CLAMSCANENGINE185

LISTING 65: BEISPIEL WEBAPP – TAGLIBRARY187

LISTING 66: BEISPIEL WEBAPP - TAG-LIBRARY-DESKRIPTOR188

LISTING 67: BEISPIEL WEBAPP – SERVLET (HELLO WORLD)190

LISTING 68: BEISPIEL WEBAPP – WEB.XML190

LISTING 69: BEISPIEL WEBAPP – KONFIGURATIONSDATEI.....191

LISTING 70: BEISPIEL WEBAPP – IMPLEMENTIERUNG WEBAPP192

LISTING 71: BEISPIEL WEBAPP – IMPLEMENTIERUNG WEBAPPCONFIGURATION197

LISTING 72: BEISPIEL WEBAPP – IMPLEMENTIERUNG WEBAPPMODULE198

LISTING 73: BEISPIEL WEBAPP – MODUL-DESKRIPTOR199

LISTING 74: FSM-ARCHIV-ERZEUGUNG – BUILD.XML203

LISTING 75: SIGNIEREN VON MODULEN MIT EINER ANT-TASK203

LISTING 76: INTERNATIONALISIERUNG MODULERESOURCES.JAVA.....207

LISTING 77: BEISPIEL GOM-FORM210

5 Klassen-, Interface-Beschreibungen

CLASS 1: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANSERVICEIMPL160

CLASS 2: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANSERVICECONFIGURATION.....171

CLASS 3: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANSERVICECONFIGPANEL172

CLASS 4: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.VSCANFILTERPROXY180

CLASS 5: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.ENGINES.CLAMAV.CLAMSCANENGINE182



INTERFACE 1: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN:VSCANSERVICE.....160
 INTERFACE 2: DE.ESPIRIT.FIRSTSPIRIT.OPT.VSCAN.SCANENGINE.....179

6 Abbildungsverzeichnis

ABBILDUNG 2-1: MODULE/KOMPONENTEN VERZEICHNISSTRUKTUR.....15
ABBILDUNG 2-2: FSM-ARCHIV-VERZEICHNISSTRUKTUR.....17
ABBILDUNG 2-3: MODUL-CLASSLOADING-HIERARCHIE.....28
ABBILDUNG 2-4: MODUL-CLASSLOADING-HIERARCHIE – CLASSLOADER-AUFTEILUNG.....29
ABBILDUNG 3-1: MODUL-BEISPIEL BUILD-VERZEICHNISSTRUKTUR.....55
ABBILDUNG 3-2: SCHREIBEN VON REFERENZEN.....110
ABBILDUNG 3-3: LESEN VON REFERENZEN.....112
ABBILDUNG 3-4: JAVA WEB START SECURITY WARNING-DIALOG.....122
ABBILDUNG 3-5: JAVA WEB START ZERTIFIKAT-CACHE / IMPORT VON ZERTIFIKATEN.....123
ABBILDUNG 3-6: SETZEN DER MODULRECHTE.....124
ABBILDUNG 3-7: ANLEGEN DER VERZEICHNISSTRUKTUR FÜR DAS JDBC-LIBRARY-MODUL.....150
ABBILDUNG 3-8: KOMPONENTENLOSES MODUL – INSTALLATION.....151
ABBILDUNG 3-9: VERWENDUNG EINES MODUL-JDBC-TREIBERS (MYSQL) – MYSQL-CONNECTOR-JAVA-3.1.14-BIN.JAR.....152
ABBILDUNG 3-10: VERWENDUNG EINES MODUL-JDBC-TREIBERS (MYSQL) – MYSQL-CONNECTOR-JAVA-3.1.14-BIN.JAR.....152
ABBILDUNG 3-11: INSTALLIERTE JDBC-MODUL-TREIBER-BIBLIOTHEK.....154
ABBILDUNG 3-12: KOMPONENTEN-TYPEN DES VSCAN-MODULS IN DER FIRSTSPIRIT SERVER- UND PROJEKTKONFIGURATION.....155
ABBILDUNG 3-13: SERVICE-KONFIGURATIONSOBERFLÄCHE IN DER FIRSTSPIRIT SERVER- UND PROJEKTKONFIGURATION.....156
ABBILDUNG 3-14: KAPSELUNG VON MODUL-KOMPONENTEN INNERHALB DER FIRSTSPIRIT SERVER- UND PROJEKTKONFIGURATION SOWIE KOMMUNIKATIONSWEGE DER KOMPONENTEN UND ANBINDUNG AN DIE EXTERNE ICAP-ANWENDUNG.....157
ABBILDUNG 3-15: VSCANSERVICECONFIGPANEL – DIE MODUL-KONFIGURATIONSOBERFLÄCHE.....172
ABBILDUNG 3-16: BEISPIEL WEBAPP – VERZEICHNISSTRUKTUR FSM.....200
ABBILDUNG 3-17: NAMENSGLEICHE MODUL-RESSOURCEN.....201
ABBILDUNG 3-18: LADEN NAMENSGLEICHER MODUL-RESSOURCEN.....202
ABBILDUNG 3-19: MODULANSICHT DES INSTALLIERTEN SIMPLE EDITOR EXAMPLE MODULS IN DER SERVER -UND PROJEKTKONFIGURATION.....210
ABBILDUNG 3-20: MÖGLICHE, FÜR DAS SIMPLE EDITOR EXAMPLE MODUL AUSFÜHRBARE AKTIONEN.....210
ABBILDUNG 3-21: MÖGLICHE, FÜR DAS SIMPLE EDITOR EXAMPLE MODUL AUSFÜHRBARE AKTIONEN IN DER FIRSTSPIRIT VERSION 4.1.....211
ABBILDUNG 3-22: DEFINITION DES FORMULARS FÜR DAS SIMPLE EDITOR EXAMPLE MODUL.....212
ABBILDUNG 3-23: DEFINITION DER HTML-DARSTELLUNG (AUSGABEKANAL) FÜR DAS SIMPLE EDITOR EXAMPLE MODUL.....212



**ABBILDUNG 3-24: DEFINITION DES PDF-AUSGABEKANALS MITTELS FORMATTING OBJECTS
PROCESSOR INNERHALB DER FIRSTSPIRIT-TEMPLATE-SYNTAX.....212**



7 Referenzen

- [1] Developer-API, <http://www.FirstSpirit.de/>, Copyright e-Spirit AG
- [2] Access-API, <http://www.FirstSpirit.de/>, Copyright e-Spirit AG
- [3] Dokumentation für Administratoren, <http://www.FirstSpirit.de/>, Copyright e-Spirit AG
- [4] Online Dokumentation für FirstSpirit – ODFS, <http://www.FirstSpirit.de/>, Copyright e-Spirit AG
- [5] Default Policy Implementation and Policy File Syntax, <http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html>, Copyright Sun Microsystems, Inc. All Rights Reserved.
- [6] Java Security, <http://java.sun.com/javase/6/docs/technotes/guides/deployment/deployment-guide/security.html>, Copyright Sun Microsystems, Inc. All Rights Reserved.
- [7] Java™ 2 Platform Security Architecture, <http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html>, Copyright © 1997-2002 Sun Microsystems, Inc. All Rights Reserved.
- [8] Java™ Web Start version 6 Frequently Asked Questions, <http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/faq.html>, March 2006
- [9] Java™ Network Launching Protocol (JNLP) Specification ("Specification"), <http://jcp.org/en/jsr/detail?id=56>, Copyright 2005 Sun Microsystems, Inc. All rights reserved.
- [10] FirstSpirit Modul-Beispiele auf Basis von Eclipse .classpath, Copyright e-Spirit AG



Logdateien	32	P	
Logging	32	Persistenz	
M		Konfigurationsdatei	185
Modul		Public-Classes (Schnittstellen für eine	
Komponente	8	Implementierung)	49
Module-Descriptor	39	S	
<components>	40, 41	Scope	
<module>	40, 203	Project	25
<name>	40, 203	projekt-lokale	8
<version>	40, 203	Server	25
Drei-Komponenten-Descriptor	158	systemweite	8
Optionale Einträge	40	Sichtbarkeit <i>Sichtbarkeit von Komponenten</i>	
Pflichtfelder	40	Web	25
Tags u. Attribute	40	web-lokale	8
Modul-Ereignisbehandlung	14	Signieren	203
<class>	14	T	
Abhängigkeiten		Typisierung und Mapping	
<depends>	14	Direkte Verwendung (Kindelement)	60
Modulrechte	124	Mengenwertige Verwendung	60
FirstSpirit Security-Manager/Classloader	124	W	
Modul-Verzeichnisstruktur	14	Webserver	
Datenverzeichnisse	14	<u>Anwendungsbeispiel</u>	22
Konfigurationsverzeichnisse	14	Extern	22
Logverzeichnisse	14	Generic	22
N		Intern	22
Namensgleichheit bei Modul-Ressourcen	11, 201		

