2 Komponenten

»Das Ganze ist mehr als die Summe seiner Teile.«

—Aristoteles, griechischer Philosoph

Der Aufbau eines Eclipse-RCP-Clients wird, wie im ersten Kapitel beschrieben, im Wesentlichen durch die Komponentenorientierung der zugrunde liegenden OSGi-Plattform bestimmt. Alle Bestandteile eines Eclipse-RCP-Clients sind demzufolge Komponenten. Dies hat nachvollziehbarerweise elementare Auswirkungen auf die Gestaltung, also die Struktur, des Clients. Dieses Kapitel beschäftigt sich mit dem Komponentenbegriff eines Eclipse-RCP-Clients.

Logische Gliederung

Die Komponentenorientierung bedingt eine logische Gliederung der einzelnen Softwareteile, um Komponenten überhaupt identifizieren zu können. Zu Beginn liefere ich Ihnen ein Schema, um unterschiedliche Typen von Komponenten anhand des Inhalts definieren zu können. Anhand dieser Typen lassen sich strukturelle Komponenten definieren. Mit zusätzlichen Strukturierungsprinzipien lassen diese sich dann zu einer gut zu verwaltenden Hierarchie zusammenfügen.

Die Komponentenstruktur beinhaltet grundsätzlich mehrere Aspekte. Zum einen ist der Schnitt der jeweiligen Komponenten, aus denen der Client zusammengesetzt ist, von Bedeutung. Der Abschnitt *Komponentenbildung* beschreibt, wie Sie diesen Schnitt schichtenübergreifend vornehmen können und somit auch für eine stringente Namensgebung der Komponenten sorgen können. Weiterhin muss über die innere Gliederung der Komponenten, also das Packaging, nachgedacht werden. Der Abschnitt *Packagestruktur* beschäftigt sich mit der internen Struktur einer Komponente und liefert Ihnen eine Empfehlung für die Strukturierung und Namensgebung der Bestandteile einer Komponente. Den Abschluss bildet das Thema Class-Loading, das im OSGi-Kontext eine besondere Bedeutung hat. Der Abschnitt zeigt Ihnen, wie Sie trotz der bestehenden Herausforderungen Ihren Eclipse-RCP-Client komponentenorientiert entwerfen können.

2.1 Komponenten in Eclipse

2.1.1 Technische Komponenten

Obwohl der Aufbau eines Eclipse-RCP-Clients als bekannt vorausgesetzt wird, sollen die einzelnen technischen Komponenten nochmals kurz eingeführt werden. Dies soll ein gemeinsames Verständnis sicherstellen.

Plug-in Ein *Plug-in* beschreibt eine Komponente des Clients. Diese Komponente beinhaltet sowohl Logik in Form von Code als auch Ressourcen in Form von Dateien, wie Bild- oder Properties-Dateien.

Fragment Ein *Fragment* ist ein OSGi-Mechanismus und bezeichnet eine Art Zusatzkomponente eines Plug-ins. Es erweitert den Inhalt eines anderen Plug-ins zur Laufzeit. Diese Komponente hat unterschiedliche Einsatzmöglichkeiten und beinhaltet je nach Verwendung entweder Code oder Dateien. Fragments werden typischerweise entweder für betriebssystemabhängigen Code, Patches, Mehrsprachigkeit oder Testklassen verwendet.

Feature Ein *Feature* stellt eine Menge von Plug-ins und/oder Fragments dar. Diese Menge steht in einem sinnvollen Zusammenhang, beispielsweise der Bearbeitung einer fachlichen Aufgabe wie Buchungsmanagement. Features sind darüber hinaus Gliederungsebenen, denen wir zu einem späteren Zeitpunkt im Rahmen des klassischen Update Managers und Java Web Start wieder begegnen werden, da hier auf Ebene der Features gearbeitet wird.

Im Kontext von Eclipse ist das Plug-in zentrales technisches Strukturelement für den Code. Fragments werden hauptsächlich verwendet, um Mehrsprachigkeit zu implementieren, wenn auch andere Verwendungszwecke denkbar und möglich sind.

2.1.2 Der Begriff Bundle

OSGi-Komponente

Seit der Version 3.0 setzt die Eclipse-Plattform auf der OSGi-Plattform auf. Damit ist eine einzelne Komponente der Eclipse-Plattform, also sowohl ein Plug-in als auch ein Fragment, gleichzeitig auch eine OSGi-Komponente, ein sogenanntes *Bundle*. Diesem Umstand ist auch zu verdanken, dass der deklarative Teil eines Eclipse-Plug-ins oder Fragments aus zwei Artefakten besteht. Der OSGi-spezifische Teil der statischen Informationen befindet sich im sogenannten *Bundle Manifest*, also der `MANIFEST.MF`-Datei. Die Eclipse-Plattform bietet darüber hinaus noch den *Extension Point*-Mechanismus. Dieser basiert ebenfalls auf

deklarativen Informationen, die im sogenannten *Plug-in Manifest*, also der `plugin.xml`, beschrieben werden. Die Begriffe Bundle und Plug-in werden im Folgenden synonym verwendet.

Die Differenzierung zwischen deklarativem und implementierendem Teil eines Bundles beweist sich im Enterprise-Umfeld als sehr vorteilhaft. Enterprise-Eclipse-RCP-Anwendungen sind typischerweise groß. Der Eclipse-RCP-Client beinhalten also viele Komponenten und viele Klassen. Da zum Startup der Anwendung die Interpretation des deklarativen Teils oft genügt, wirkt sich dies positiv auf die Startzeit der Anwendung aus. Die entsprechende Implementierung wird dann jeweils bei Bedarf geladen.

Deklarativer und implementierender Teil

2.1.3 Bundletypen

Plug-in, Fragment und auch Bundle repräsentieren rein technische Aspekte zur Komponentenbildung und stellen die Grundlage der Entwicklung dar. Für die Bildung von Komponenten empfiehlt es sich, die Komponenten auch inhaltlich zu kategorisieren.

Auf Ebene der Bundles lassen sich unterschiedliche Typen anhand des Inhalts identifizieren. Die aufgeführten Typen stellen dabei keine offizielle Definition dar, sie sollen vielmehr Kategorien zur Unterscheidung einzelner Bundles bilden. Die Unterscheidung ist für die Architektur des Clients insofern von Relevanz, als dass die identifizierten Typen von Bundles fachliche bzw. technische Komponenten der Software darstellen. Dies ist sehr wichtig, da ein Eclipse-RCP-Client im Allgemeinen aus vielen Bundles besteht. Die einzelnen Komponenten sollten also so entworfen werden, dass sie sinnvoll geschnitten und austauschbar sind für den Fall, dass Teile der Software verändert werden.

Kategorien zur Unterscheidung

Code Bundle Ein *Code Bundle* beschreibt ein Standard-Plug-in des Clients. Diese Komponente beinhaltet sowohl Logik in Form von Code als auch Ressourcen in Form von Bild- oder Properties-Dateien.

Library Bundle Ein *Library Bundle* definiert ein Plug-in des Clients, das eine 3rd-Party-Bibliothek kapselt. Diese Komponente beinhaltet ausschließlich den Code der Bibliothek.

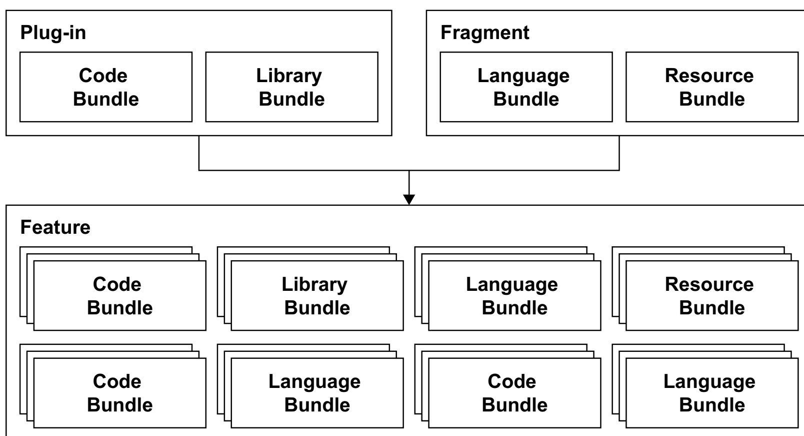
Resource Bundle Ein *Resource Bundle* stellt den typischen Anwendungsfall eines Fragments dar. Das Resource Bundle beinhaltet im Normalfall keinen Code, sondern lediglich Konfigurationen, spezifische Build- oder Properties-Dateien.

Language Bundle Ein *Language Bundle* ist ein spezieller Typ Resource Bundle, also ebenfalls ein Fragment. Es beinhaltet eine Properties-Datei mit Übersetzungen für Code Bundles.

2.1.4 Bundles und Features

Der Begriff Feature findet auf Ebene der Eclipse Bundles kein Äquivalent. Er dient weiterhin als Gruppierungselement, das wir uns im Zusammenhang mit dem Update Manager später genauer anschauen werden. Features haben allerdings keinen Einfluss auf die tatsächliche Struktur des Codes. Abbildung 2-1 zeigt den Zusammenhang der Begriffe Plug-in, Fragment und Feature zu den Typen von Eclipse Bundles und stellt deren Beziehung zueinander dar.

Abb. 2-1 *Clientseitige EERCP-Komponenten*



Sowohl für Library als auch für Language Bundles ergibt sich ein relativ offensichtlicher Rahmen für den Inhalt, da dieser bereits durch den Bundletyp vorgegeben ist. Bei Code Bundles gestaltet sich der richtige *Schnitt* deutlich anspruchsvoller, da hier der Inhalt wiederum komponentenorientiert gestaltet werden sollte.

2.2 Strukturierungsprinzipien

Mit der Strukturierung des Clients sollte das Ziel verfolgt werden, möglichst leicht zu verwaltende Komponenten zu entwerfen. Wichtig ist dabei die funktionale Abgeschlossenheit des Inhalts der Komponenten, da es gerade das Abhängigkeitsmanagement deutlich vereinfacht. Der Inhalt kann dabei sowohl nach fachlichen als auch nach technischen Gesichtspunkten unterschieden werden. Im Folgenden soll die Strukturierung der Code Bundles betrachtet werden.

2.2.1 Fachliche Bundles: Der richtige Schnitt

Use Cases

Da die Entwicklung des Clients im Normalfall anhand einzelner Anwendungsfälle, sogenannten *Use Cases*, erfolgt, empfiehlt es sich, die Code Bundles des Clients auch anhand der Use Cases zu strukturieren. Dieses Vorgehen sichert zwei wesentliche Vorteile. Zum einen bilden Use Cases per Definition eine sinnvolle, in sich abgeschlossene Menge an Geschäftslogik. So kann die bereits erfolgte Komponentenbildung der Designphase für den Client weiterverwendet werden. Damit sind Funktionalität und Inhalt in sich abgeschlossen, fachliche Abhängigkeiten zu anderen Use Cases bzw. Komponenten sind bereits vollständig definiert. Gegebenenfalls ist es sinnvoll, zwei oder mehrere Use Cases zusammenzufassen, wenn sie thematisch eine Einheit bilden. Des Weiteren kann sich dieses Vorgehen später für die Autorisierung auszahlen, da diese meist auf Basis von Use Cases und Rollen basiert.

2.2.2 Technische Bundles: Querschnittsfunktionalität

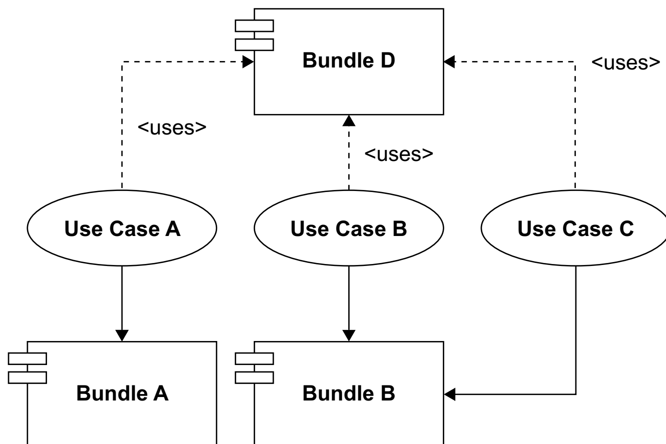
Strukturiert man die Code Bundles anhand der fachlichen Komponenten, dann taucht schnell Funktionalität auf, die übergreifend benötigt wird. Auch Querschnittsfunktionalität sollte sinnvoll in einzelne Komponenten aufgeteilt werden. So kann man zum Beispiel technische Komponenten, die Funktionen wie Databinding oder spezielle Widgets beinhalten, definieren. Diese Funktionen stehen typischerweise in keinem direkten Zusammenhang zu einem bestimmten fachlichen Aspekt. Darüber hinaus gibt es die fachlichen Querschnittsfunktionen wie Labeling, Content Assist für bestimmte Entitäten oder Jobs, die man als Komponente wiederverwendbar bereitstellen möchte. Jede Funktionalität muss dahingehend überprüft werden, ob sie als eigenes Code Bundle oder als Teil eines anderen Code Bundles zur Verfügung gestellt wird. Für größere Funktionalitäten wie beispielsweise Databinding oder Security ist eine eigene Komponente zu empfehlen, das Releasemanagement wird so deutlich vereinfacht. Kleinere Funktionalitäten und einzelne Widgets hingegen sollten in einer Sammelkomponente zur Verfügung gestellt werden. Es ist sehr schwer, Richtwerte für Anzahl, Größe und Umfang der von Ihnen entwickelten Komponenten zu nennen. Entscheidend ist eher, wie gut und sauber Sie diese inhaltlich gestalten. Im Allgemeinen gilt der Grundsatz *»So viele wie nötig, so wenige wie möglich«*.

Kein direkter Zusammenhang zu einem bestimmten fachlichen Aspekt

Abbildung 2-2 auf der nächsten Seite zeigt den Zusammenhang zwischen Use Cases und Komponenten. Use Case A wird in Komponente A abgebildet. Use Case B und C gehen in Komponente B auf.

Komponente D bildet die Querschnittsfunktionalität aller drei Use Cases ab.

Abb. 2-2
Komponentenschnitt



2.3 Komponentenbildung

Die Bildung von Komponenten beinhaltet zwei Aspekte, die äußere und die innere Struktur. Mit der äußeren Struktur ist die Art der Aufteilung der Anwendung in Bundles gemeint, die innere Struktur beschreibt den inneren Aufbau eines Bundles selbst.

2.3.1 Bundle-Struktur

Mit der Unterscheidung der unterschiedlichen Typen von Bundles ist bereits eine Strukturierungsmöglichkeit geschaffen. Der Code eines Clients ist in der Regel sehr umfangreich, wobei der Anteil von selbst geschriebenem Code, also Code Bundles, im Vergleich zu den verwendeten Bibliotheken, also Library Bundles, überwiegen wird.

Bei der Gestaltung Ihrer Komponenten existieren gegebenenfalls firmeninterne Konventionen, die Sie berücksichtigen müssen. Dennoch möchte ich an dieser Stelle ein Standardlayout für Ihre Code Bundles empfehlen. Diese Struktur hat sich bewährt und erlaubt eine flexible Handhabung Ihrer Enterprise-Eclipse-RCP-Anwendung.

Namensgebung

Der symbolische Name eines jeden Bundles beginnt also mit der First- und Second-Level-Domäne des Eigners der zu entwickelnden Anwendung. Daran schließt sich der Name der Anwendung an. Unter der Voraussetzung, dass Sie Ihre Bundles entsprechend Ihrer Use-Case-Definitionen gliedern, ist zu empfehlen, als Nächstes den Namen der Komponente zu verwenden, zu dem das Bundle gehört. Zum einen

ist das Best Practice der Eclipse-Plattform. Zum anderen setzt sich die Komponente meist aus zwei Teilen, den clientseitigen Elementen und den serverseitigen Elementen, zusammen. Das Code Bundle beinhaltet dabei lediglich die UI-Teile der Komponente, die aber das gleiche Präfix in der Packagestruktur tragen wie der serverseitige Teil der Komponente. Für den UI-Teil einer Komponente, also die Elemente des UI Code Bundles, würde dementsprechend folgendes Pattern für die Namensgebung zur Anwendung kommen:

Einen UI-Teil benennen

***Domäne.Firma.Anwendung.Komponente*.ui**

In unserem Beispiel heißt der UI-Teil der Personalverwaltungskomponente also:

```
net.sf.dysis.resource.ui
```

Dieses Code Bundle würde entsprechend alle UI-Elemente bezüglich der Personalverwaltung beinhalten. Abbildung 2-3 zeigt die Übersicht der zum Dysis-Client gehörenden Bundles, die entsprechend strukturiert sind.

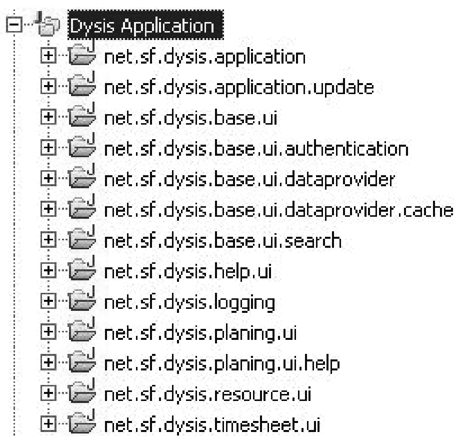


Abb. 2-3
Überblick über die Bundles des Dysis-Client

Für den Fall, dass Sie serverseitig ebenfalls mit OSGi arbeiten, können die serverseitigen Bundles nach demselben Prinzip mit einem anderen Suffix benannt werden:

***Domäne.Firma.Anwendung.Komponente*.core**

Der Core-Teil der Personalverwaltungskomponente in unserem Beispiel hieße also:

Einen Core-Teil benennen

```
net.sf.dysis.resource.core
```

Dieses Bundle würde alle Domänenobjekte, Services und die Geschäftslogik der Komponente beinhalten. Denkbar wären dann auch Utility Code Bundles, die spezifische Querschnittsfunktionen beinhalten. Das Auslagern in ein eigenes Bundle hat den Vorteil, dass Sie das Bundle dann sowohl auf der Clientseite als auch auf der Serverseite verwenden können.

Sollten Sie serverseitig nicht mit OSGi arbeiten, dann sollten Sie dennoch den für das serverseitige Bundle vorgeschlagenen symbolischen Namen als Präfix im Packagenamen verwenden. Sie erreichen so auf Packageebene eine sehr gute Zuordnung von client- und serverseitigem Code.

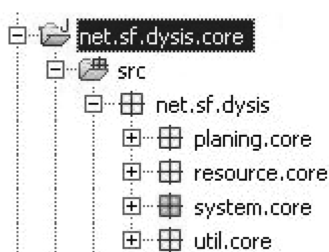


Abb. 2-4 *Überblick über die Aufteilung der Packages des Dysis-Servers*

2.3.2 Packagestruktur

Nachdem wir einen Blick auf die Bildung von Komponenten geworfen haben, schauen wir uns nun die interne Struktur eines Code Bundles, also das Packaging, an. Grundsätzlich bleibt es natürlich Ihnen überlassen, wie Sie die Packagestruktur gestalten. Eventuell existieren auch hier interne Konventionen, denen Sie entsprechen müssen. Allerdings ist es zu empfehlen, vorher eine bindende Vorgabe zu definieren, um eine homogene Struktur komponentenübergreifend sicherzustellen. Meine Empfehlung hat sich im praktischen Einsatz bewährt und stellt ebendieses sicher.

Bindende Vorgabe definieren

Für das Naming von Eclipse Bundles gelten grundsätzlich die Regeln, die auch bei der Entwicklung von Plug-ins gelten. Das bedeutet, das *Root Package* entspricht dem Namen des Plug-ins.

Clientseitige Packages

In einem Code Bundle hat es sich bewährt, die Packagestruktur eher technisch zu gestalten. Unterpackages wie *.editor*, oder *.dialog* beinhalten die UI-Elemente, die direkt zur Komponente gehören. Für den Fall, dass Sie Schnittstellen oder Funktionalität einer anderen Komponente bzw. eines anderen Bundles verwenden, empfiehlt es sich, die Klassen in

einem eigenen Package zu verwalten. Beispiel hierfür wäre ein Package *.databinding* oder *.search*.

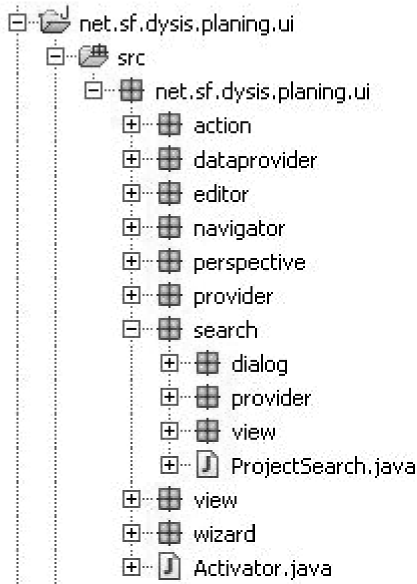


Abb. 2-5
Ein Bundle gliedert sich intern in die einzelnen technischen Elemente.

Serverseitige Packages

Für die Serverseite ist das gleiche Vorgehen zu empfehlen. Hier fänden sich typischerweise Unterpackages wie *.domain*, das das Domain Model beinhaltet, oder *.service* mit den vom UI verwendeten Services.

Abb. 2-6
Die Packages unterhalb der Komponenten gliedern sich im Dysis-Serverprojekt in die jeweiligen technischen Aufgabenbereiche.

2.3.3 Public und Private API

Bundles definieren nicht nur Abhängigkeiten auf andere Bundles, sondern exportieren zusätzlich explizit Elemente. Dieser Export geschieht auf Packageebene. Exportierte Packages stellen das sogenannte *Public API*, nicht exportierte Packages das *Private API* dar. Die Definition, welches Package sichtbar ist und welches nicht, findet sich im Bundle Manifest und ist auf den ersten Blick nicht ersichtlich. Es hat sich bei

Bundles, die Packages exportieren, als praktisch erwiesen, das *Private API* mit einem Zusatz *internal* in der Packagebezeichnung zu kennzeichnen. Exportiert wird dann grundsätzlich alles, was kein *internal* im Packagenamen trägt.

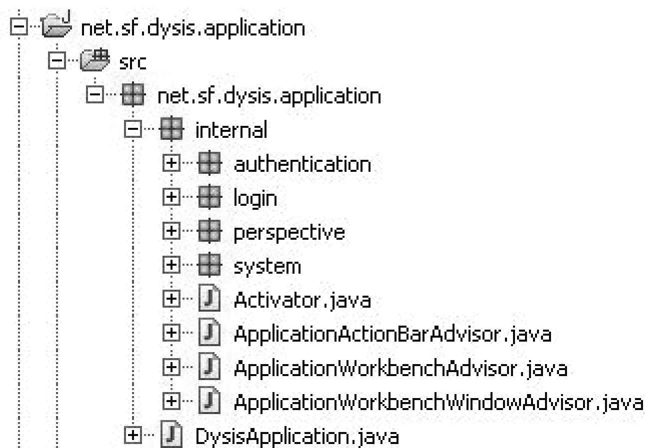


Abb. 2-7 *Das Private API des Bundles net.sf.dysis.application befindet sich im internal Package.*

2.4 Versionierung von Bundles

Wir haben über unterschiedliche Arten von Bundletypen gesprochen. Dann haben wir uns Beispiele für einzelne Bundles anhand eines Standardlayouts angeschaut. Die Einteilung in einzelne Komponenten steht dabei im Vordergrund. Im Zuge dieser Komponentenbildung kommt dann ein zweiter wesentlicher Aspekt der Entwicklung von Software hinzu, die Versionierung. Da Eclipse auf OSGi basiert, wird die Versionierung bereits von der Plattform nativ unterstützt. Jedes einzelne Bundle wird zusätzlich zum Namen durch eine Versionsinformation auf dem OSGi-Bus identifiziert, die Kombination ist eindeutig.

2.4.1 Versionierungsschemata

Wie auch bei der Nomenklatur ist für die Versionierung ein Schema empfehlenswert. Zunächst ist zu definieren, wie die Versionen der einzelnen Bundles inkrementiert werden. Dabei ist zu betonen, dass sich dieses Schema lediglich auf Code und Resource Bundles bezieht. Library Bundles sollten grundsätzlich die Version der beinhalteten externen Bibliothek als Version führen.

Individuell versionieren

Eine separate Versionierung der einzelnen Bundles ermöglicht es, die Entwicklung der Komponenten spezifisch zu verfolgen. Dies führt dazu, dass die ausgelieferte Anwendung fachliche Komponenten in unterschiedlichen Versionen beinhaltet. Die Version der Anwendung wird im Feature der Anwendung und in dem Bundle, das die Produktdefinition enthält, geführt. Dieses Vorgehen hat den Vorteil, dass bei einem Versionsupdate der Anwendung tatsächlich nur die Komponenten aktualisiert werden müssen, die modifiziert bzw. erweitert wurden.

Bundles haben unterschiedliche Versionsnummern.

Einheitlich versionieren

Im Gegensatz zur spezifischen Versionierung kann auch die gesamte Anwendung, also alle fachlichen Komponenten, mit einer einheitlichen aktuellen Versionsnummer ausgeliefert werden. Diese Versionsnummer ist dann identisch mit der des Features. Zwar werden so bei einem Versionsupdate immer alle fachlichen Komponenten der Anwendung aktualisiert, was zu zusätzlichem Datenverkehr führt. Allerdings ist einer Komponente damit direkt anzusehen, zu welcher Version sie gehört, und sie ist damit unabhängig vom Feature zuzuordnen.

Bundles haben identische Versionsnummern.

Bewertung

Bisher habe ich für Eclipse-RCP-Anwendungen immer ein einheitliches Versionierungsschema eingesetzt. Durch das automatische Inkrementieren aller Komponenten nach einem Release ist sichergestellt, dass sämtliche Änderungen definitiv im kommenden Release enthalten sind. Darüber hinaus ist es sehr komfortabel, auf den ersten Blick zu wissen, zu welcher Version der Anwendung die jeweilige Komponente gehört. Da ein Versionsupdate tendenziell immer einen längeren Prozess darstellt, kann damit der zusätzliche Datenverkehr entschuldigt werden. Darüber hinaus wird in einigen Fällen ohnehin nicht der interne Update Manager verwendet, beispielsweise bei Einsatz einer Softwareverteilung. Sollten Sie allerdings ausschließlich den Eclipse Update Manager einsetzen und viele einzelne verteilte Clients betreiben, dann kann genau der Datentransfer ein Grund für eine individuelle Versionierung darstellen.

2.4.2 Versionsnummern vergeben

Die Versionsnummer eines Bundle besteht typischerweise aus mindestens drei ganzen Zahlen *Major*-, *Minor*- und *Micro*-Version, welche durch jeweils einen Punkt getrennt werden. Üblicherweise wird dann

Versionsnummer: major.minor.micro.qualifier

noch ein alphanumerischer *Qualifier* als Suffix hinzugefügt. Die einzelnen Teile der Versionsnummer können für den gesamten Client sehr gut dafür verwendet werden, um die Kompatibilität mit dem verbundenen Backend zu bewerten.

Unabhängig davon, ob Sie Ihre Bundles einheitlich oder individuell versionieren, können Sie die Version der Anwendung zur Kompatibilitätsprüfung verwenden.

Major-Version Ein Inkrementieren der Major-Version bedeutet eine elementare Veränderung der Anwendung. Dies kann beispielsweise neue Funktionalität oder strukturelle Modifikationen am Datenmodell beinhalten. Ein Backend mit einer anderen Major-Version ist nicht kompatibel mit dem Client.

Minor-Version Ein Inkrementieren der Minor-Version bedeutet eine strukturelle Modifikation der Anwendung. Diese Modifikation macht das Backend in der Regel inkompatibel mit dem Client. Es gibt allerdings auch den Ansatz, für eine Minor-Version-Inkrementierung lediglich zusätzliches API zu erlauben und den Bruch des bestehenden APIs zu verbieten. So wäre der Client dann weiterhin kompatibel mit dem Server.

Micro-Version Die Micro-Version sollte dann inkrementiert werden, wenn eine Modifikation keine strukturellen Auswirkungen hat und ein älterer Client der gleichen Minor-Version trotz der Veränderung weiterhin mit dem Backend zusammenarbeiten kann. Dies beinhaltet beispielsweise Änderungen oder Erweiterungen der Geschäftslogik.

Qualifier Der Qualifier bietet zusätzliche Information für die ausgelieferte Version. So ist es beispielsweise für Nightly Builds üblich, einen Zeitstempel hinzuzufügen. Der PDE-Build bietet hierfür auch eine integrierte Unterstützung.

Die Vergabe der Versionsnummer nach dem beschriebenen Schema macht es möglich, gerade in der Entwicklungszeit ggf. auf eine Aktualisierung des Clients verzichten zu können. Sollten lediglich Bestandteile der Geschäftslogik manipuliert worden sein, dann ist es so möglich, diese weiterhin mit dem älteren Client zu testen.

2.4.3 Verschiedene Versionen des gleichen Bundles verwenden

Der Einsatz von unterschiedlichen Versionen des gleichen Bundles findet in den allermeisten Fällen bei Library Bundles Anwendung. Es kann durchaus vorkommen, dass zwei Bundles verschiedene Versionen ei-

nes Library Bundles benötigen. Dies kann beispielsweise bei Logging-Frameworks oder XML-Parsern der Fall sein. Wichtig ist hierbei, dass auf die Komponentenhierarchie geachtet wird, um Konflikte durch zyklische Abhängigkeiten zu vermeiden. Innerhalb fachlicher Komponenten sollte versucht werden, mit einer gemeinsamen Version zu arbeiten. Behält man allerdings die Komponentenhierarchie unter Kontrolle, dann ist auch der Einsatz verschiedener Versionen des gleichen Bundles problemlos möglich.

2.5 Bundle Classloading

Die Verwendung von Bundles bringt auf technischer Ebene eine wesentliche Veränderung mit sich. Bundles haben grundsätzlich ihren eigenen Classloader und damit einen separaten Klassenpfad. Dieser – zugegeben sehr technische – Aspekt wirkt sich indirekt auf die Art und Weise aus, wie Komponenten gestaltet werden. Die strikte Trennung der Klassenpfade bedingt, dass die Abhängigkeiten zwischen einzelnen Komponenten respektive Bundles genau durchdacht sein müssen.

2.5.1 Standard-Classloading

Java verwendet für die im Code genutzten Klassen einen Classloader. Dieser ist dafür zuständig, den Bytecode der Klassen in den Ressourcen auf dem Klassenpfad zu finden und zu laden, damit die Klasse entsprechend genutzt werden kann. Die geladene Klasse wird dann zukünftig über ihren Namen und den Classloader, der für die Bereitstellung verwendet wurde, in der Java VM identifiziert. Potenziell sieht Java es also vor, eine Klasse mehrere Male pro VM laden zu können. Diese Funktion wird in Java-Programmen jedoch üblicherweise nicht bewusst verwendet. Lediglich Applikationsserver nutzen die Classloader-Hierarchie, um die Klassenpfade der einzelnen installierten Anwendungen voneinander zu trennen. Innerhalb einer Anwendung wird allerdings gewöhnlich ein einziger Classloader verwendet, es existiert somit lediglich ein einziger Klassenpfad.

Identifikation von Klassen in der Java VM

2.5.2 Classloader Hell

Die Tatsache, dass innerhalb einer Anwendung ein einziger Classloader mit einem einzigen Klassenpfad verwendet wird, hat das Design einiger Bibliotheken sehr beeinflusst. Dies betrifft insbesondere Bibliotheken, die Reflection verwenden, um auf bestimmte Ressourcen zuzugreifen. Diese Bibliotheken arbeiten meist mit Beschreibungsdateien, die Strukturen oder Konfigurationen für eine Anwendung definieren. Dabei wer-

Reflection

den häufig anwendungsspezifische Klassen mit Hilfe von Reflection erzeugt und verwendet.

Bidirektionale Beziehungen

Die Verwendung solcher Bibliotheken setzt voraus, dass einerseits der Teil der Anwendung auf die Klassen der Bibliothek zurückgreifen kann. Das scheint offensichtlich, um die Bibliothek überhaupt verwenden zu können. Die Bibliothek wiederum benötigt ebenfalls Zugriff auf die Klassen der Anwendung, die mittels Reflection erzeugt werden sollen.

Um Bibliotheken möglichst effizient managen zu können, ist entsprechend des beschriebenen Komponentenmodells die Auslagerung in Library Bundles der Weg der Wahl. Da Library Bundles allerdings aufgrund der zyklischen Abhängigkeit keinerlei Abhängigkeiten zu anwendungspezifischen Bundles haben können, ist die notwendige Bidirektionalität problematisch. In der Praxis besteht aber die Notwendigkeit einer Lösung, um entsprechende Bibliotheken einsetzen zu können.

Buddy Classloading

Erweiterung des Klassenpfads

Eclipse Equinox, die OSGi-Implementierung von Eclipse, stellt das Buddy Classloading zur Verfügung, um eine bidirektionale Abhängigkeit für den Classloader verwaltbar zu gestalten. Buddy Classloading macht es möglich, den Klassenpfad eines Bundles zu erweitern. Sollte ein Bundle Classloader über seinen Klassenpfad die gesuchte Klasse nicht finden, dann fragt er bei den registrierten Bundles, den sogenannten *Buddys*, nach. Um in einem Bundle das Buddy Classloading zu erlauben, wird im Bundle Manifest der Eintrag »Eclipse-BuddyPolicy: *strategy name*« vorgenommen.

Es gibt unterschiedliche Strategien, wie Buddys ermittelt werden. Eine solche Strategie nennt sich Buddy Policy und wird, wie in Abbildung 2-12 auf Seite 47 zu sehen, im Bundle Manifest definiert.

registered Der Buddy-Mechanismus wird bei allen Bundles nachfragen, die sich als Buddy für das Bundle registriert haben. Bundles, die sich bei einem solchen Bundle als Buddy registrieren möchten, führen im eigenen Manifest die Registrierung durch den Eintrag »Eclipse-RegisterBuddy: *bundleSymbolicName*« durch, wie in Abbildung 2-13 auf Seite 47 zu sehen. Wichtig ist dabei, dass das als Buddy registrierte Bundle zusätzlich eine Dependency auf das Bundle definiert.

dependent Diese Strategie bewirkt, dass der Buddy-Mechanismus in allen Bundles nachschaut, die eine Abhängigkeit auf dieses Bundle

besitzen. Dabei ist nicht relevant, ob das abhängige Plug-in dies erlaubt.

global Die gesuchte Klasse wird hier in der gesamten Menge an exportierten Paketen aller bekannten Bundles gesucht.

app Der Buddy-Mechanismus delegiert an den Application Classloader, der den gesamten Klassenpfad durchsucht.

ext Der Extension Classloader sucht im Extension-Verzeichnis (bspw. `<JAVA_HOME>/lib/ext`) das Verzeichnis des JDK bzw. der JRE.

boot Die Klasse wird mit dem Boot Classloader in allen originären JDK bzw. JRE Libraries gesucht.

2.5.3 Beispiel

Die Problematik des Classloadings möchte ich an einem Beispiel verdeutlichen. Es gibt viele Bibliotheken oder Frameworks, die dafür verwendet werden können, so unter anderem Hibernate, Log4J oder das Spring Framework. Für dieses Beispiel verwende ich das Spring Framework. Sollten Sie nicht mit Spring vertraut sein, dann ist das an dieser Stelle nicht schlimm. Die für das Beispiel relevante Funktionalität wird zu Beginn erläutert. Darüber hinaus werde ich jedoch keine detaillierte Beschreibung über Funktion und Arbeitsweise des Spring Frameworks liefern, sondern auf einen für das Classloading relevanten Punkt eingehen. Spring eignet sich als Beispiel zudem zur Beschreibung einer komponentenorientierten Lösung auf Basis von OSGi. Diese wird vom Spring Framework selbst in Form des Spring-Dynamic-Modules-Projekts bereitgestellt.

Erläuterung anhand des Spring Frameworks

Das Spring Framework

Das Spring Framework verwendet einen sogenannten *Spring Application Context*, in dem einzelne in der Anwendung verwendete Objekte, sogenannte *Spring Beans*, aufgeführt sind. Für den Spring Application Context liegen gewöhnlich eine oder mehrere Kontext-XML-Dateien vor, die vom Spring Framework zu Beginn eingelesen werden.

Spring Beans

Listing 2.1
Die Kontext-XML-Datei

```
<beans>

  ...

  <bean id="requestExecutor" class="net.sf.dysis.core.client.spring
      .HttpInvokerRequestExecutor" />
```

```
<bean id="serviceExceptionHandlingAdvice" class="net.sf.dysis.
    core.client.internal.spring.ServiceExceptionHandlingAdvice"/>

...

</beans>
```

Dependency Injection

Der Spring Application Context ermöglicht es, mit Dependency Injection (DI) zu arbeiten. Das Spring Framework sorgt dabei zum einen für die Erzeugung der aufgeführten Spring Beans durch den Java-Reflection-Mechanismus. Darüber hinaus werden die Beziehungen der Spring Beans zueinander aufgelöst, indem entsprechend der Konfiguration einer Spring Bean andere Beans als Attribut injiziert werden.

Anwendungsszenario

Clientseitiger Spring Application Context

Die Referenzimplementierung *Dysis* verwendet für die Kommunikation zwischen Client und Backend das Spring Remoting. Dabei existiert clientseitig ein Spring Application Context, der eine injektive Abbildung bestimmter Services des Backends darstellt. Die im clientseitigen Application Context aufgeführten Service Proxys mappen dabei genau einen Service der Serverseite. Der clientseitige Aufruf wird vom Spring Framework transparent an das Backend delegiert, welches dann die Ausführung des Aufrufs gewährleistet und das Ergebnis zurückliefert.

Schnittstellen-Bundle

Die im Client benötigten Klassen, also Service Interfaces und DTOs, werden als eigenes Bundle *net.sf.dysis.core.client* im Client zur Verfügung gestellt. Dieses Bundle enthält ebenfalls die Kontext-XML-Datei zur Beschreibung des Spring Application Context. Dem beschriebenen Komponentenmodell entsprechend würde das Spring Framework als Library Bundle eingebunden werden. Das *net.sf.dysis.core.client* Bundle mit den Klassen der Serverschnittstelle würde eine Abhängigkeit auf das Spring Library Bundle definieren, da es einige Klassen des Spring Frameworks verwendet. Da die Kontext-XML-Dateien allerdings im Bundle *net.sf.dysis.core.client* liegen, benötigt das Spring Library Bundle eine Abhängigkeit auf das Bundle, um auf die Kontext-XML-Dateien und die in ihnen definierten Klassen zugreifen zu können.

Abhängigkeiten zur Spring-Bibliothek

Lösungsstrategien

Im Fall Spring existieren drei unterschiedliche Lösungsstrategien, wobei davon zwei die strikte Trennung der Komponenten aufgeben.

Kopien der Bibliotheken verwenden Die wohl einfachste, aber auch schlechteste Möglichkeit besteht darin, komplett auf das Spring Library

Bundle zu verzichten und die Spring-Bibliotheken direkt in das Dysis Core Client Bundle zu kopieren.

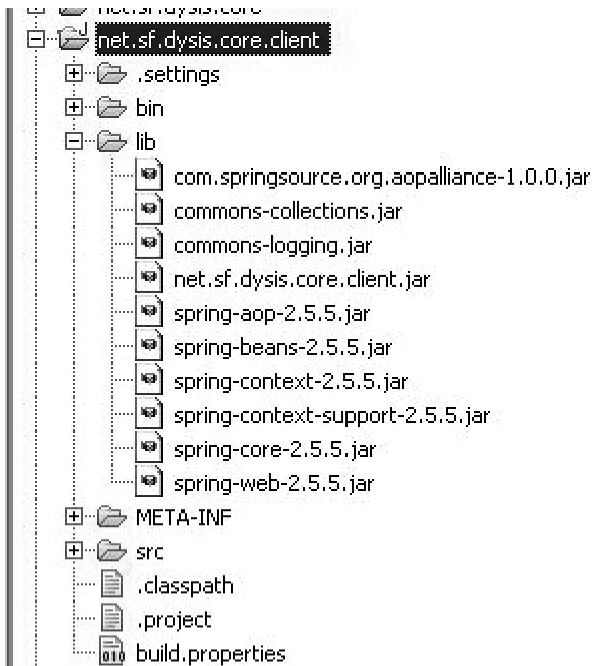


Abb. 2-8
In der ersten Variante liegen die benötigten Spring-Bibliotheken integriert im Bundle. Sie sind damit Bestandteil des Bundles und können nicht separat verwaltet werden.

Die Spring-Bibliotheken liegen also im Dysis Bundle selbst, wie in Abbildung 2-8 zu sehen. Über das Bundle Manifest werden die Bibliotheken dann dem Klassenpfad des Dysis Plug-ins zur Verfügung gestellt (siehe Abbildung 2-9 auf der nächsten Seite). Der Spring Application Context, die Klassen der Serverschnittstelle und die Spring-Bibliotheken liegen somit automatisch im selben Klassenpfad.

Diese Lösung ist, wie gesagt, die schlechtestmögliche. Sie stellt einen massiven Bruch der Komponentenorientierung dar, da damit das Spring Framework und dessen benötigte Bibliotheken elementarer Bestandteil des Dysis Bundles werden und nicht mehr separat verwaltet werden können.

Buddy Classloading Als zweite Möglichkeit kann das Buddy Classloading verwendet werden. Die Spring-Bibliotheken sind dabei in einem Library Bundle gekapselt, welches die benötigten Klassen bereitstellt. Abbildung 2-10 zeigt die Runtime-Konfiguration des Spring Library Bundles.

Das Library Bundle exportiert die in den gekapselten Bibliotheken enthaltenen Klassen. Um die vom Library Bundle bereitgestellten Klassen nutzen zu können, definiert das Dysis Bundle eine Abhängigkeit auf

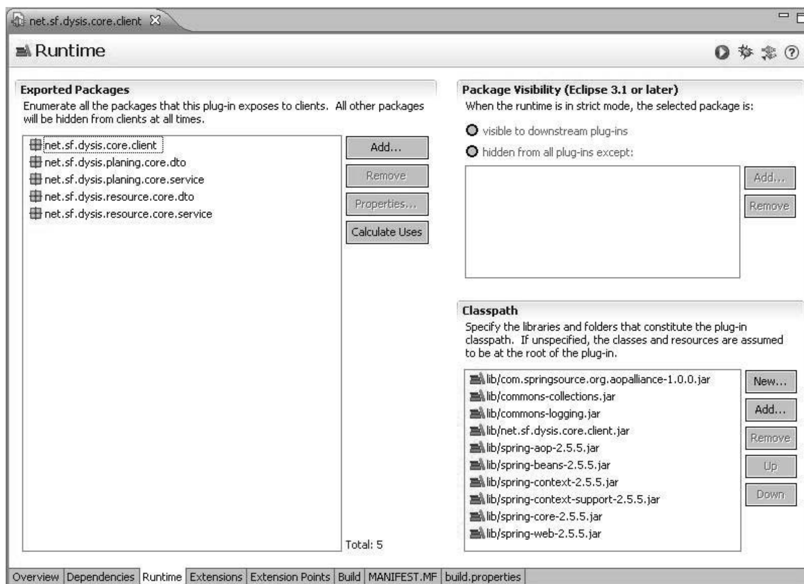


Abb. 2-9
Die im Bundle enthaltenen Spring-Bibliotheken werden dem Klassenpfad im MANIFEST.MF des Dysis Bundles hinzugefügt. Damit kann das Bundle zum einen mit den Klassen arbeiten, zum anderen haben die integrierten Klassen Zugriff auf die Klassen des Bundles.

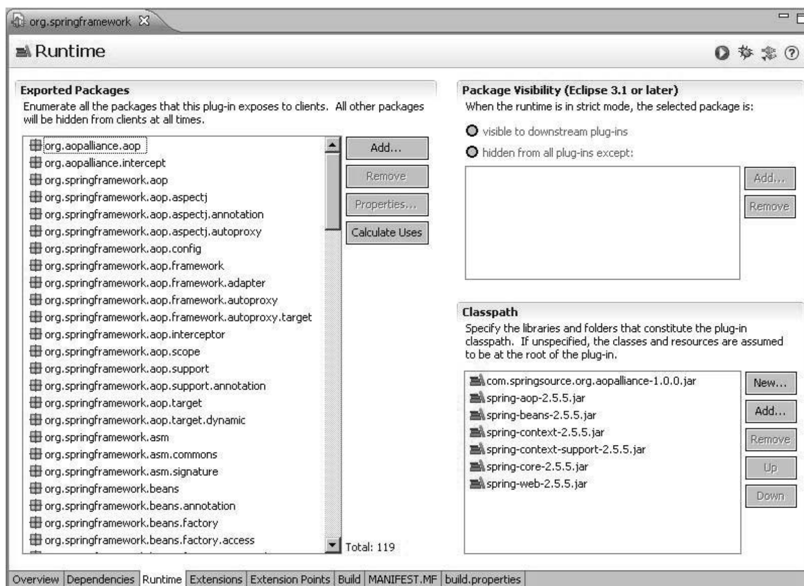


Abb. 2-10
Die Spring-Bibliotheken liegen in der zweiten Variante gekapselt in einem eigenen Library Bundle und exportieren die entsprechend benötigten Packages. So kann die Bibliothek separat verwaltet werden.

das Spring Library Bundle, wie man in Abbildung 2-11 auf der nächsten Seite sehen kann.

Der Zugriff, den das Spring Library Bundles auf die im Dysis Bundle liegenden Klassen beim Starten des Spring Application Context benötigt, wird mit Hilfe des Buddy Classloading gelöst. Dazu wird, wie in Abbildung 2-12 auf der nächsten Seite zu sehen, der für eine *registered*

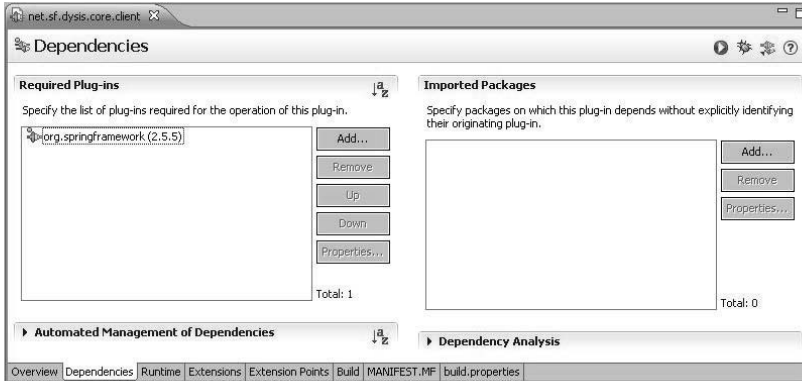


Abb. 2-11
Das Spring Library Bundle wird dem Dysis Bundle im MANIFEST.MF als Dependency hinzugefügt.

Buddy Policy notwendige Eintrag »Eclipse-BuddyPolicy: *registered*« im Bundle Manifest des Spring Library Bundles vorgenommen.

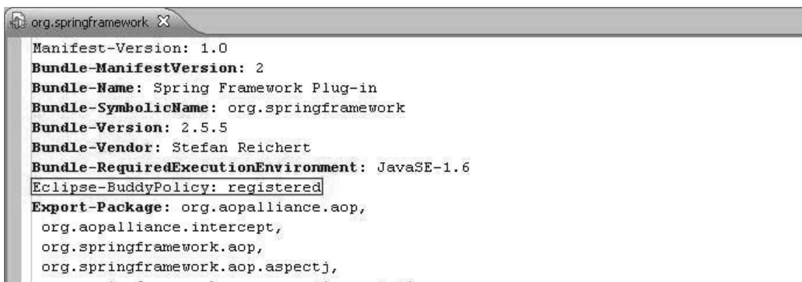


Abb. 2-12
Die Buddy Policy wird im MANIFEST.MF des Bundles hinterlegt.

Das Dysis Bundle registriert sich als Buddy für das Spring Library Bundle, indem es im Bundle Manifest den Eintrag »Eclipse-RegisterBuddy: *org.springframework*« erhält, wie in Abbildung 2-13 dargestellt ist.

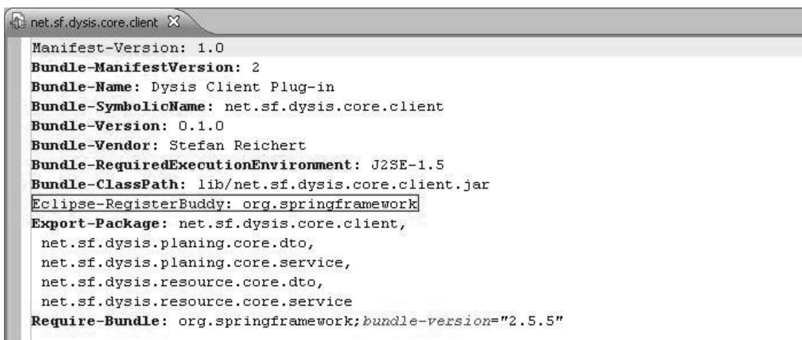


Abb. 2-13
Die Registrierung als Buddy für ein anderes Bundle mit der »register« Buddy Policy wird im MANIFEST.MF des Bundles hinterlegt.

Spring Dynamic Modules Das Spring Framework bietet als dritte Möglichkeit mit dem Spring-Dynamic-Modules-Projekt (Spring DM)[1] eine Verteilung des Application Context über OSGi Bundles an. Dabei nutzt Spring DM den OSGi-Bundle-Kontext, um die in anderen Bundles vorhandenen Kontext-XML-Dateien zu finden und für diese Bundles einen Spring Application Context mit Hilfe der Kontext-XML-Datei(en) des Bundles zu erstellen.

Spring DM stellt hierfür drei zusätzliche Bundles zur Verfügung, die für das Aufspüren und Erstellen von einem Spring Application Context für ein anderes Bundle zuständig sind.

Spring DM Bundles

- org.springframework.bundle.osgi.core
- org.springframework.bundle.osgi.extender
- org.springframework.bundle.osgi.io

Der OSGi-Philosophie entsprechend wird dann auch nicht ein einziges Spring Library Bundle für die eigentliche Spring-Bibliothek verwendet. Die Spring DM Distribution enthält für die notwendigen Module des Spring Frameworks jeweils einzelne eigenständige Bundles:

Spring Bundles

- org.springframework.bundle.spring.aop
- org.springframework.bundle.spring.beans
- org.springframework.bundle.spring.context
- org.springframework.bundle.spring.context.support
- org.springframework.bundle.spring.core
- org.springframework.bundle.spring.web
- com.springsource.org.aopalliance

Sobald diese Bundles eingebunden sind, werden Kontext-XML-Dateien im Ordner *META-INF/spring* automatisch erkannt und für einen Spring Application Context verwendet.

Das Interessante ist nun, dass durch Spring DM die Bidirektionalität in den Abhängigkeiten aufgelöst wird. Lediglich das Dysis Core Client Bundle mit der Serverschnittstelle benötigt jetzt noch eine Abhängigkeit auf die einzelnen Bundles der Module des Spring Frameworks. Das Spring Framework selbst nutzt dank Spring DM den OSGi-Kontext, um Zugriff auf die entsprechenden Klassen zu erhalten.

Spring stellt für das beschriebene Classloading-Problem einen Sonderfall dar, da es selbst für eine OSGi-basierte Lösung sorgt. In den meisten Fällen werden Sie mit Hilfe von Buddy Classloading aber wenigstens

[1] Spring DM arbeitet daran, den OSGi Blueprint Service (RFC 124) zu implementieren und hierfür zur Referenzimplementierung zu werden.

eine stringente Komponentenstruktur schaffen können, auch wenn so genau genommen die gewünschte strikte Trennung der Klassenpfade einzelner Komponenten aufgehoben wird. Gänzlich abzuraten ist von der Verwendung von Kopien der Bibliotheken in den einzelnen Bundles. Sie würden so mit der eigentlich gewünschten Komponentenorientierung brechen.