akquinet

Spring & Eclipse Equinox kombiniert

Dipl.-Inform. Martin Lippert (martin.lippert@akquinet.de)

akquinet

Über mich

Martin Lippert

- Senior-IT-Berater bei Akquinet Agile GmbH
- martin.lippert@akquinet.de

Schwerpunkte:

- Agile Softwareentwicklung
- Refactoring
- Eclipse-Technologie

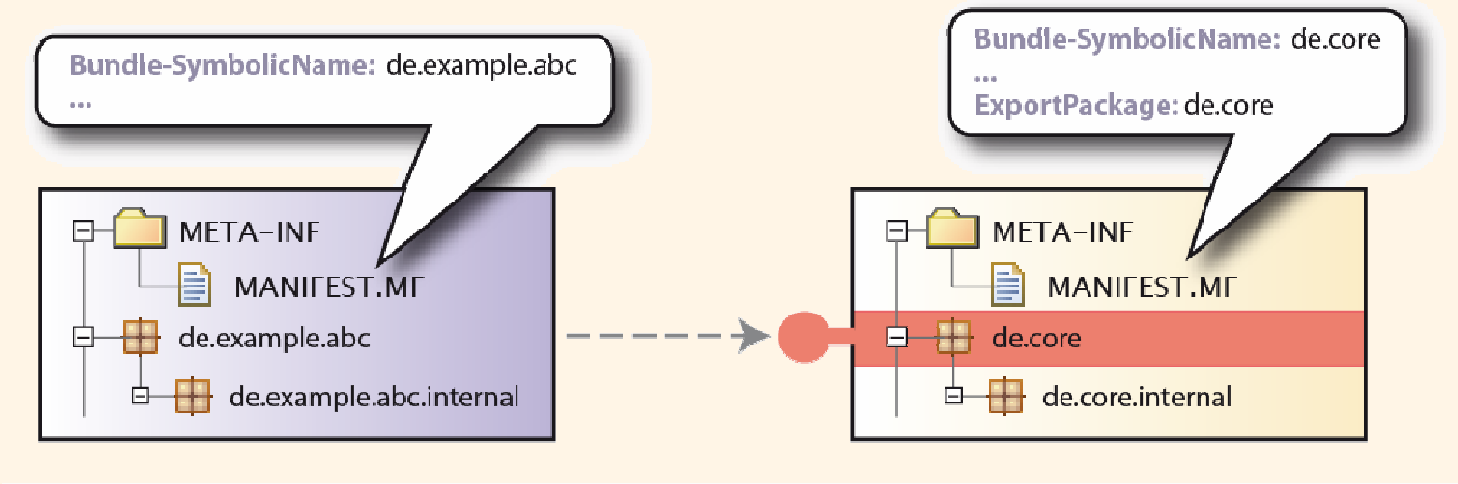
Committer im Equinox Incubator Projekt

akquinet

Inhalt

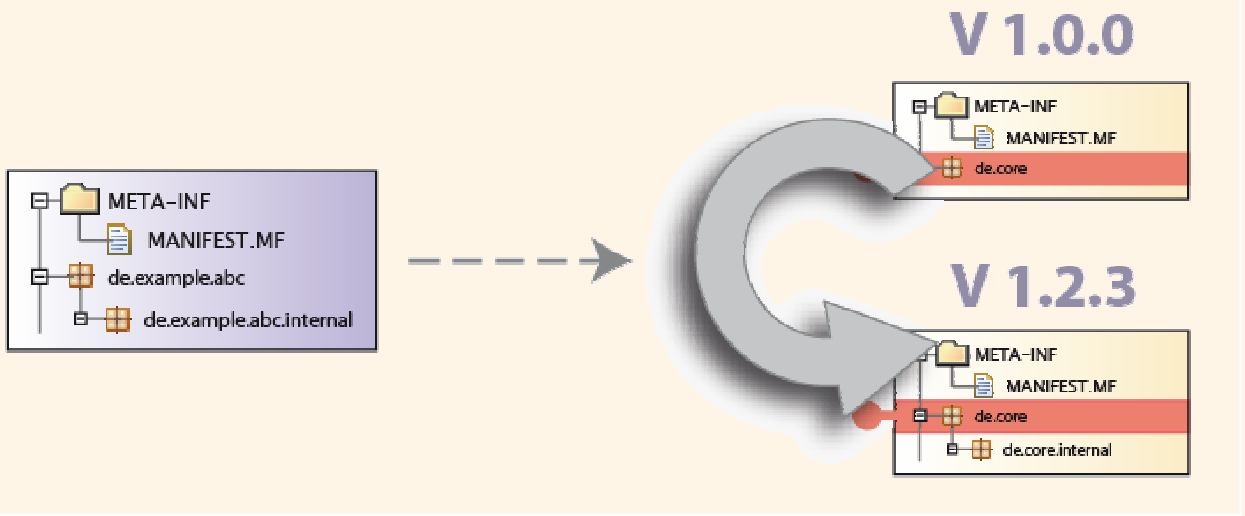
- Warum Spring & OSGi?
- Die Grundidee
- Der Stand der Dinge
- Ein praktisches Beispiel
- Fazit

Warum OSGi? – Modularisierung!



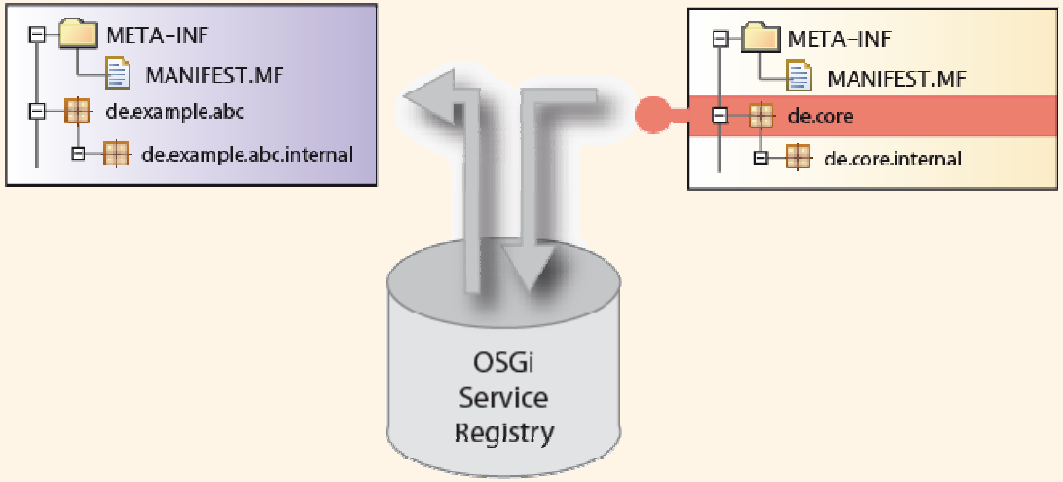
- Applikationslogik kann in Module (Bundles) aufgeteilt werden
- Bundles können eine öffentliche Schnittstelle definieren
- Bundles müssen die von ihnen benötigten Packages definieren

Warum OSGi? – Hot-Deployment!



- Bundles können zur Laufzeit installiert, aktualisiert und deinstalliert werden
- Unterschiedliche Versionen einen Bundles können gleichzeitig im System existieren

Warum OSGi? – Services!



- OSGi Service Registry enthält Services, die systemweit sichtbar sind
- Bundles können Service dynamisch an der Registry an- und abmelden
- Bundles können Services an der Registry abfragen und verwenden

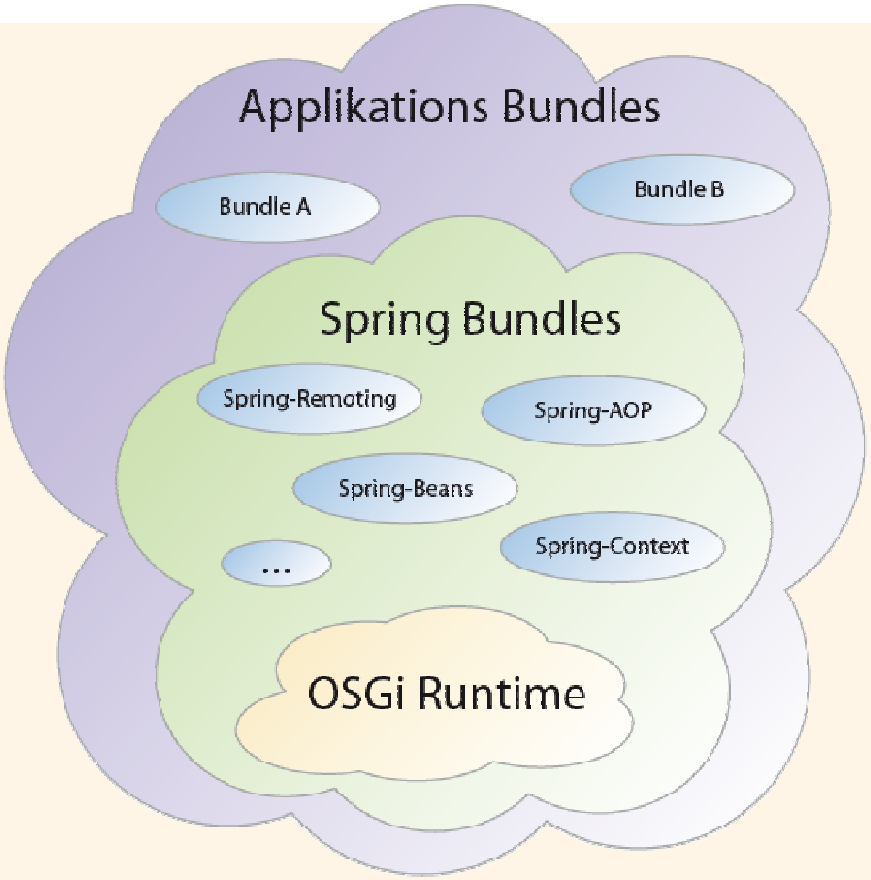
akquinet

Warum Spring?

- Einfaches Programmiermodell (POJOs)
- Einfachere Testbarkeit
- Vermeidung von Abhängigkeiten durch Dependency Injection bzw. Inversion of Control
- Integrationsplattform für unterschiedliche Technologien (AOP, Persistence, Remoting etc.)
- Vereinfachte Benutzung von Java EE und andere Java APIs

Spring & OSGi

- OSGi-Runtime als Infrastuktur
- Spring-Module als Bundles
- Anwendungslogik auf Basis von Spring als Bundles



akquinet

Wie können beide Welten kombiniert werden?

Spring:

- Beschreibung von Beans durch Application Context-Definitionen (XML)
- Typischerweise ein Application-Context pro Applikation
- Abhängigkeiten und Injektionen werden statisch definiert

OSGi:

- Mehrere Bundles innerhalb einer Applikation
- Ein Application Context pro Applikation passt nicht in die OSGi-Welt
- Die Dynamik von OSGi erscheint ebenfalls zunächst problematisch

akquinet

Eine ideale Kombination

Idee:

- Spring wird in Form von Bundles bereitgestellt
- Spring-basierte Anwendung besteht aus Bundles
- Ein Application Context pro Bundle

Spring Beans als OSGi Services:

- Spring Beans können als OSGi Services exportiert werden
- OSGi Services können als Spring Beans importiert werden
- Umgang mit dynamisch installierten und deinstallierten Services über den Standard Spring Proxy Mechanismus

akquinet

Der Stand der Dinge

- Das Spring Team arbeitet derzeit an der Spring-OSGI-Integration
- Integration ist Schwesterprojekt, parallel zu Spring 2.1

Adressen:

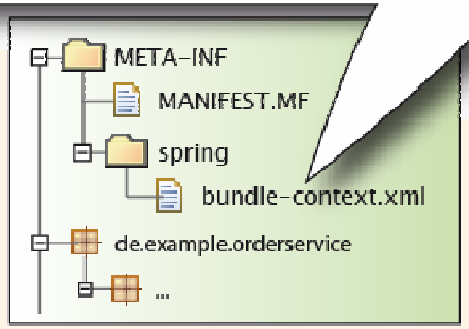
- Website unter http://www.springframework.org/osgi/
- Google Group unter http://groups.google.com/group/spring-osgi
- Alpha-Version verfügbar unter https://svn.sourceforge.net/svnroot/springframework/springosgi/trunk

Ein einfaches Beispiel

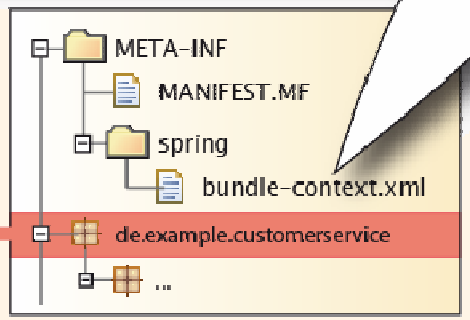
```
<bean id="orderService"
 class="de.example.orderservice.OrderService">

  <property name="customerService">
    <ref bean="customerService">
  </property>

</bean>
```



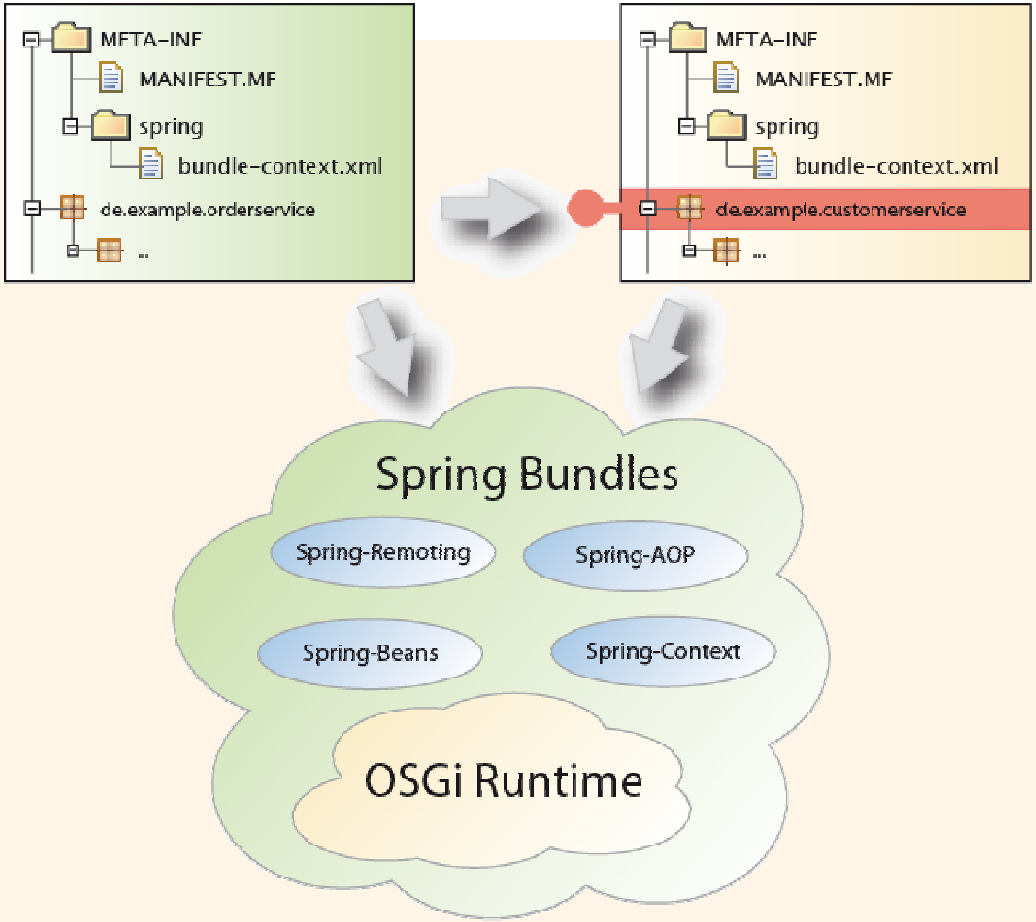

```
<bean id="customerService"
 class="de.example.customerservice.CustomerService"/>
```



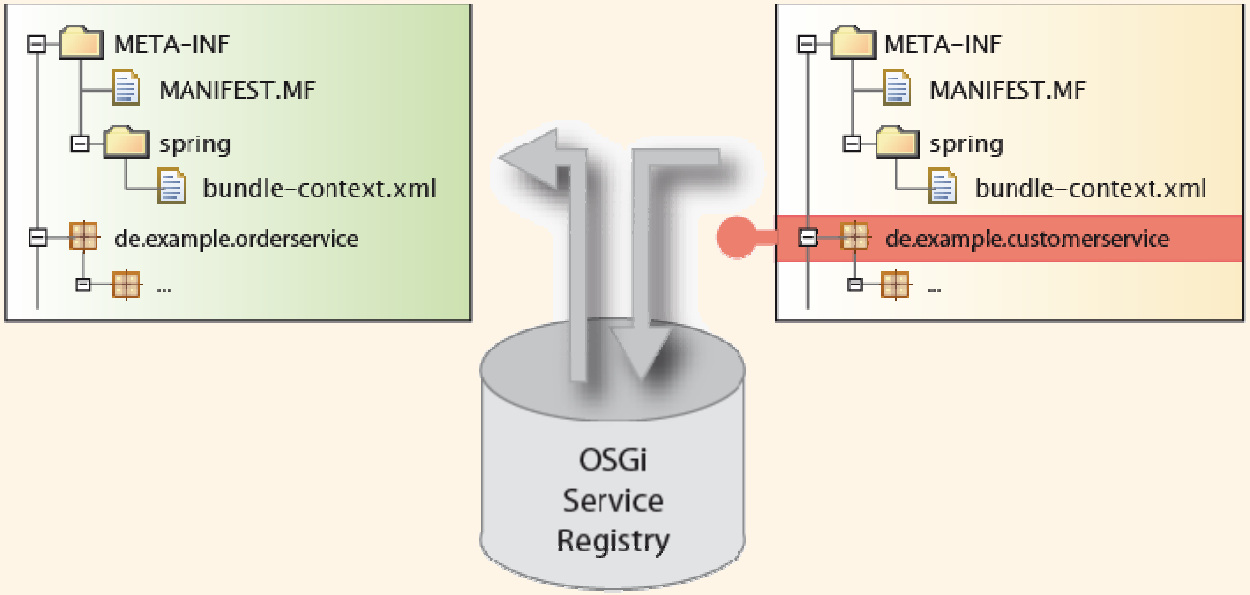
- Bundle ‚CustomerService' definiert eine Bean ‚customerService'
- Bundle ‚OrderService' definiert eine Bean ‚orderService'
- OrderService hat Referenz auf die Bean , customerService'

Realisierung

- Jedes Plugin besitzt eine eigene BeanFactory
- Plugin-übergreifende Injektion von Beans möglich
- Dynamisches Verhalten wird unterstützt



Wie werden Beans plugin-übergreifend bereitgestellt?

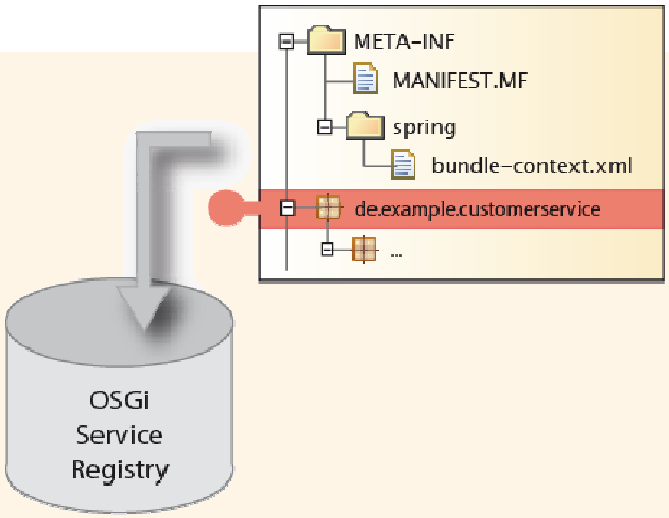


Lösung:

- Beans werden über die OSGi-Service-Registry als OSGi-Services bereitgestellt
- Zugriff über Proxy-Mechanismus möglich

Beans als OSGi-Services

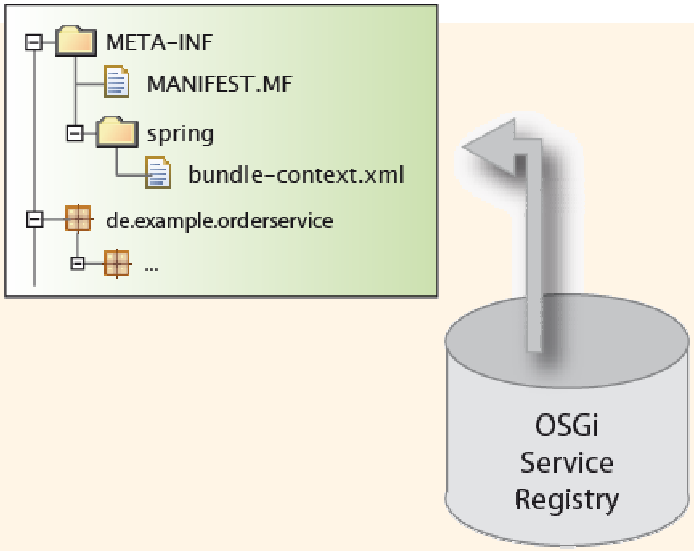
- Beans können als OSGi-Services exportiert werden
- OSGi-Services sind plugin-übergreifend sichtbar




```
<beans>
  <bean name="customerService"
        class="de.example.customerservice.internal.CustomerServiceImpl" />

  <osgi:service id="customerServiceOsgi"
                ref="customerService"
                interface="de.example.customerservice.CustomerService" />
</beans>
```

OSGi-Services als Beans



- OSGi-Services können als Beans bereitgestellt werden
- Proxy-Objekte kapseln den Zugriff auf die OSGi-Services

```
<beans>
  <osgi:reference id="customerServiceOsgi"
                  interface="de.example.customerservice.CustomerService"/>

  <bean id="orderService„
            class="de.example.orderservice.internal.OrderServiceImpl">
    <property name="customerService">
      <ref local="customerServiceOsgi"/>
    </property>
  </bean>
</beans>
```

akquinet

Herausforderungen I

Classloading:

- Spring benutzt eine Vielzahl von Open-Source-Bibliotheken
- Einige davon machen innerhalb einer OSGi-Umgebung Schwierigkeiten, in der Regel aber lösbare Probleme ☺

Dynamisches Verhalten von OSGi:

- keine Probleme mit dem Spring Proxy Mechanismus, aber:
- mögliche Probleme mit anderen Bibliotheken wie z.B. Hibernate, etc.

akquinet

Herausforderungen II

Skalierbarkeit:

- Spring-Definitionen werden bei Aktivierung des Bundles gelesen
- Was passiert, wenn es tausende von Bundles gibt?
- Müssen alle Bundles aktiviert werden oder kann dies auch erst passieren, wenn die Bundles benötigt werden?
  - Stichwort: Meta-Daten, ähnlich der Extension-Registry

akquinet

Fazit

Verbindung von OSGi und Spring bietet einige Vorteile

- Komponentenmodell mit Trennung von Aufgaben
- einfaches Programmiermodell
- einfache Einbindung von Java EE Technologien
- Dynamisches Verhalten von Applikationen

Es bleiben Herausforderungen

- dynamisches Verhalten kann zu Problemen führen
- Classloading

Fragen? Jederzeit gerne!

Email: martin.lippert@akquinet.de

Beratung, Coaching, Training

Besuchen Sie uns auf unserem Stand !

Vielen Dank an:

Gerd Wütherich (comdirect AG)

Bernd Kolb (kolbware)