OSGi mit Apache Felix und IntelliJ IDEA

Inhalt

Einführung

Die Möglichkeiten für Modularität in Java sind bescheiden. Man kann Module als JAR-Packages erstellen, jedoch ist es nicht möglich Abhängigkeiten zwischen den JARs zu definieren. Desweiteren gibt es Probleme, wenn Module Abhängigkeiten zu verschiedenen Versionen der gleichen Bibliothek haben. Auch die Sichtbarkeit (public, protected, private, package private) von Klassen und Schnittstellen kann nur low-level auf objektorientierter Sicht definiert werden. Dies ist beispielsweise problematisch, wenn mann innerhalb eines logischen Moduls den Zugriff auf eine Klasse aus einem anderem Package (auch Subpackage) benötigt. Man muss diese Klasse dann als public deklarieren. Das bedeutet aber auch, dass alle anderen Module auf diese Klasse zugreifen können, obwohl die Klasse keine öffentliche Schnittstelle darstellen sollte.

Aus diesem Schmerz heraus ist OSGi entstanden. OSGi schafft die Möglichkeit in Java logische Module zu erstellen, Abhängigkeiten zu definieren, die Module zu verwalten und den Lebenszyklus dieser zu überwachen.

Prinzipiell ist die OSGi-Spezifikation über das JSR 291 als Java-Standard angenommen. Sun Microsystems war jedoch dagegen, so dass OSGi bis heute nicht in den Java Standard integriert wurde. Im kommenden Java 8 Release soll Java endlich erweiterten Support für Modularität erhalten (JSR 337, Project Jigsaw). Wie dieser genau aussieht, ist aber noch nicht klar.

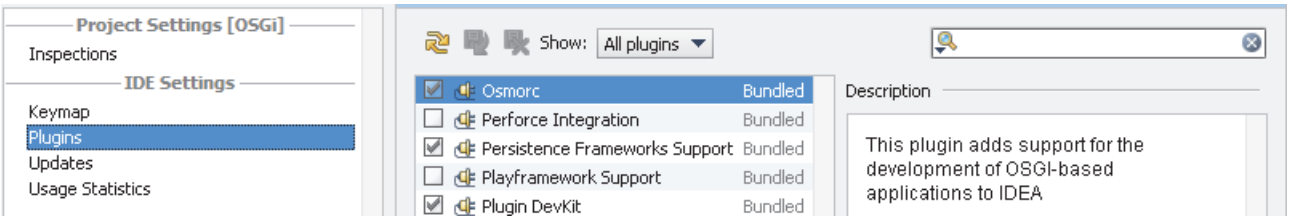
Der folgende Artikel soll eine Kurzeinführung für OSGi anhand des Apache Felix Frameworks und der Entwicklungsumgebung IntelliJ IDEA geben.

Voraussetzungen

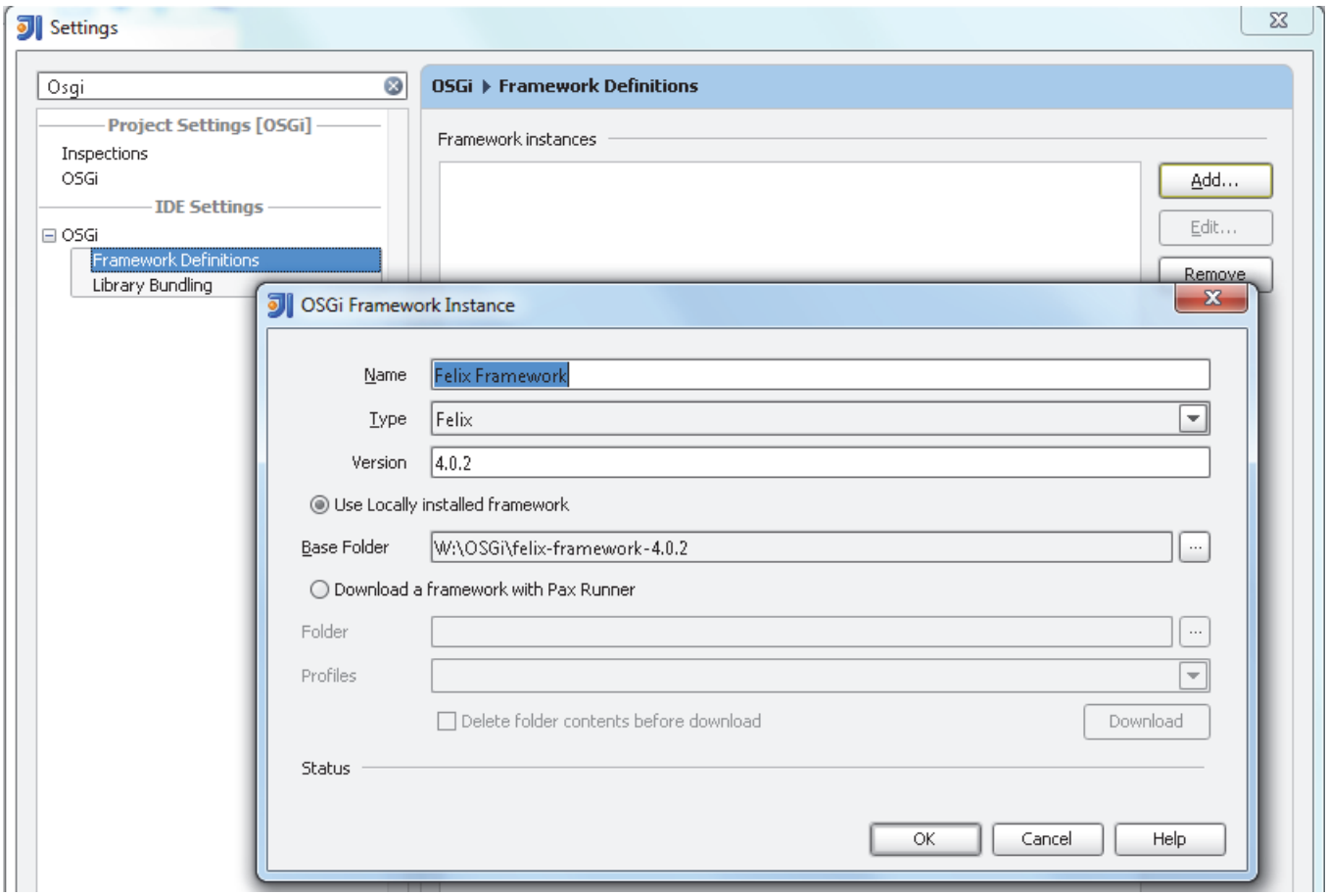
- IntelliJ IDEA 10/11 - hervorragende Entwicklungsumgebung für Java
- Apache Felix Framework 4.0.x - OSGi Framework

Apache Felix als OSGi Framework in IntelliJ IDEA konfigurieren

Als erstes muss sichergestellt werden, dass OSGi in IntelliJ IDEA aktiviert ist. Hierfür muss man den Einstellungsdialog öffnen und unter Plugins "Osmorc" aktivieren. War das Plugin vorher noch nicht aktiviert, muss die IDE neu gestartet werden, ansonsten sind die OSGi-Features nicht verfügbar.



Nun muss noch Apache Felix als OSGi Framework in IntelliJ IDEA konfiguriert werden. Dazu muss man im Einstellungsdialog unter den "IDE Settings/OSGi" den Punkt "Framework Definitions" wählen, und durch drücken von "Add" das Apache Felix Framework hinzufügen. Als Base-Folder wählt man hierfür das Verzeichnis, in welches man Apache Felix entpackt hat.

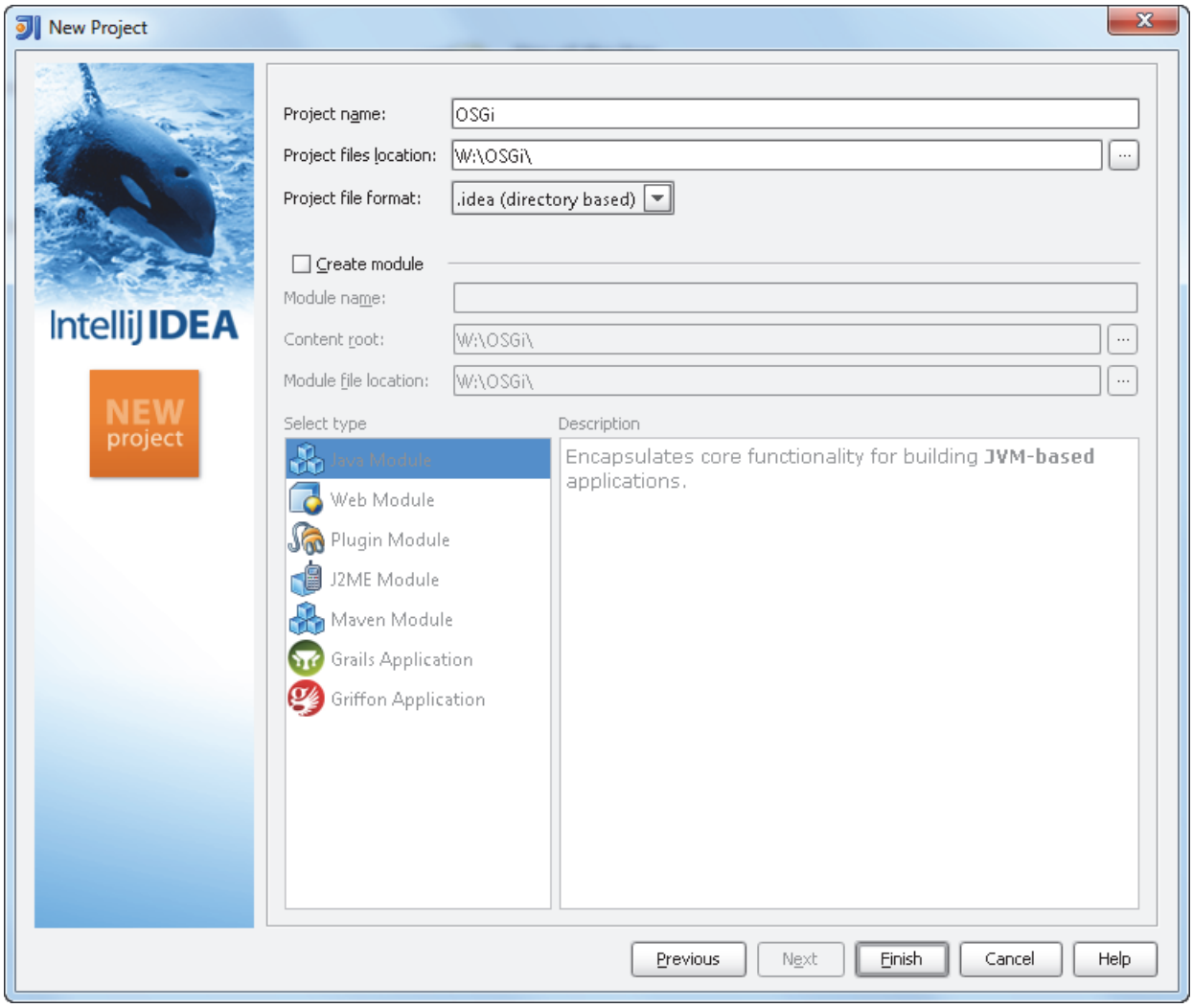


Projektstruktur in IntelliJ IDEA aufsetzen

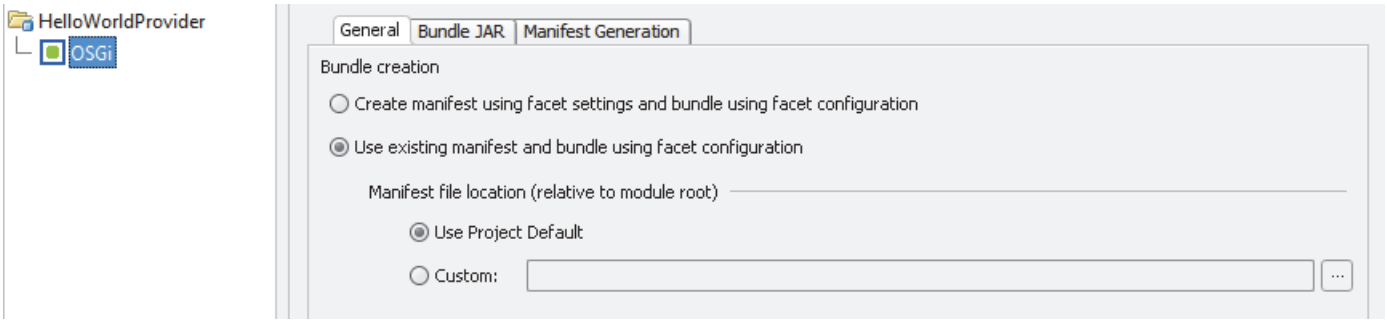
Für unser Beispiel setzen wir ein IntelliJ IDEA Projekt mit den folgenden IntelliJ Modulen auf:

- **HelloWorldProvider** - OSGi Bundle, welches den Service zur Verfügung stellt
- **HelloWorldConsumer** - OSGi Bundle, welches den Service von HellWorldProvider aufruft
- **HelloWorldClient** - IntelliJ IDEA Modul, welches die Demonstration startet

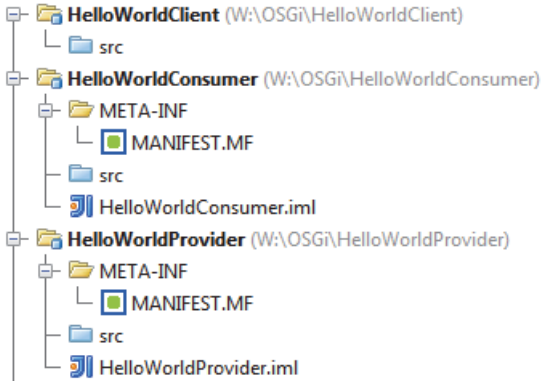
Dazu wird erst einmal ein neues Projekt ohne Modul erstellt.



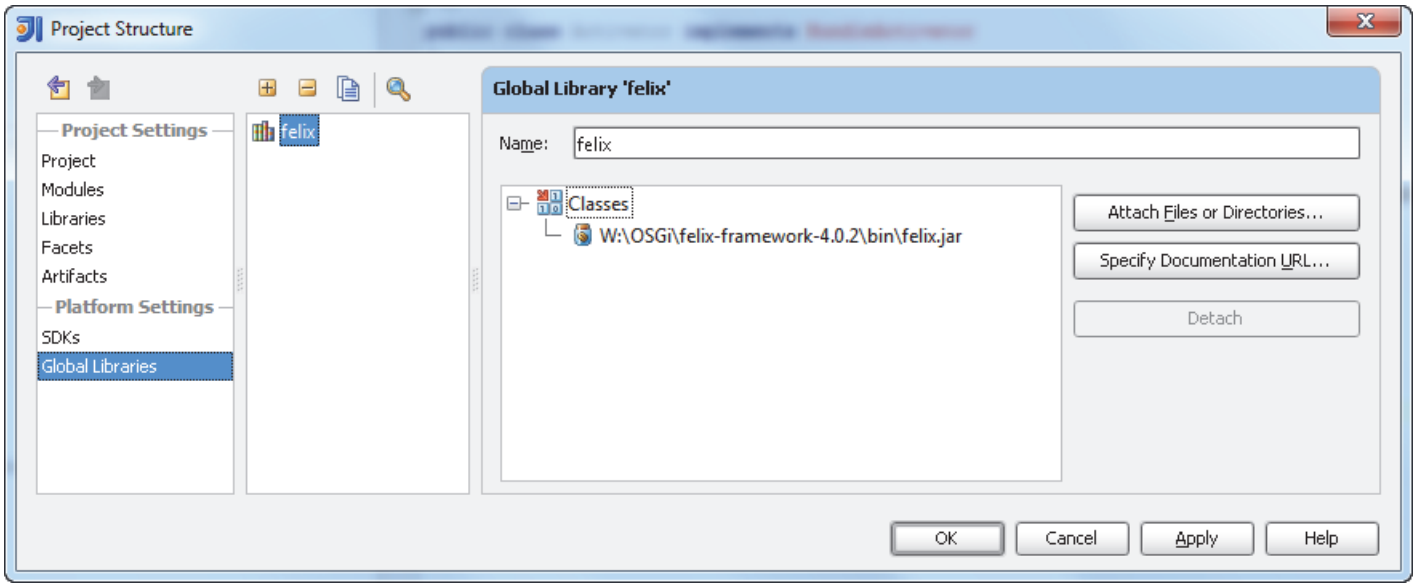
Danach erstellen wird die drei Module "HelloWorldProvider", "HelloWorldConsumer" und "HelloWorldClient". Für die beiden Module "HelloWorldProvider" und "HelloWorldConsumer" muss man noch das Facet "OSGi" hinzufügen, und jeweils im Tab "General" die Option "Use existing manifest and bundle using facet configuration" wählen. Danach erscheint ein Fehler, dass kein Manifest-File existiert. Durch drücken auf den Button "Create" wird das Manifest von IntelliJ angelegt.



Wurden die Module korrekt angelegt, sollte die Struktur wie folgt aussehen:



Damit sich die Klassen von Apache Felix im Classpath der Module befindet, muss noch eine globale Bibliothek angelegt und allen drei Modulen zugeordnet werden.



Aufbau eines Bundles

Ein logisches Modul wird in OSGi "Bundle" genannt und besteht aus einem JAR. Die OSGi Metadaten finden sich in der Manifest-Datei "META-INF/MANIFEST-MF" wieder. Somit kann man durch anreichern von Metainformationen aus einem normalen JAR ein OSGi Bundle machen.

Der Standard-Aufbau eines OSGi-Manifest sieht folgendermaßen aus:

```
Bundle-ManifestVersion: 2
Bundle-Name: Human readable name
Bundle-SymbolicName: unique bundle identifier
Bundle-Version: 1.0.0.qualifier
```

Das Attribut "Bundle-ManifestVersion" wurde aus Kompatibilitätsgründen eingeführt. Version "2" bedeutet, dass sich hier um ein OSGi-Bundle der Spezifikation R4 oder später handelt. Der "Bundle-Name" ist ein beliebiger Kurzname, ist aber für die Identifikation des Bundles irrelevant und stellt nur eine Hilfe für den Benutzer dar. OSGi verwendet zur Identifikation die Attribute "Bundle-SymbolicName" und "Bundle-Version". Das OSGi-Versionsschema sieht drei numerische Komponenten (Major, Minor, Micro) und einen optionalen alphanumerischen "qualifier" vor. Wird eine numerische Komponente nicht angegeben, wird 0 angenommen. Die Version "1" wird also als "1.0.0" aufgelöst.

HelloWorldProviderBundle

Das HelloWorldProvider Bundle bietet einen HelloWorldService an. Damit man gegebenfalls auch die Implementierung austauschen kann, wird zuerst ein HelloWorldService-Interface angelegt.

```
package org.jvcode.osgi.provider;

public interface HelloWorldService
{
    void helloWorld();
}
```

HelloWorldServicejava

Der eigentliche Service HelloWorldProvider implementiert dieses Service Interface:

```
package org.jvcode.osgi.provider;

public class HelloWorldProvider implements HelloWorldService
{
    public void helloWorld()
    {
        System.out.println("=== Hello World! ===");
    }
}
```

HelloWorldProviderjava

OSGi bietet eine Service-Registry, für welche man Dienste registrieren kann. Um den Dienst automatisch beim Starten des Bundles zu registrieren, wird ein OSGi-BundleActivator implementiert.

```
package org.jvcode.osgi.provider;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator
{
    public void start(BundleContext bundleContext) throws Exception
    {
        System.out.println("HelloWorldProvider is being started...");
        System.out.println("Registering HelloWorldService..");
        bundleContext.registerService(HelloWorldService.class.getName(),
                                      new HelloWorldProvider(), null);
    }

    public void stop(BundleContext bundleContext) throws Exception
    {
        System.out.println("HelloWorldProvider is being stopped...");
    }
}
```

Activator.java

Das BundleActivator-Interface fordert die Implementierung der beiden Methoden "start" und "stop". In der Methode start wird der Service über die Methode "registerService" des Objekts "BundleContext" registriert. Der erste Parameter gibt den Namen an, unter welcher der Service verfügbar sein soll. Als zweiter Parameter wird das eigentliche Service-Objekt übergeben. In diesem Falle wird eine neue Instanz von HelloWorldProvider erzeugt. Der dritte Parameter ermöglicht es zusätzliche Properties zu übergeben.

Damit die Activator-Klasse von OSGi aufgerufen wird und die Service-Klassen nach außen sichtbar sind, muss das Manifest erweitert werden.

```
Bundle-Activator: org.jvcode.osgi.provider.Activator
Import-Package: org.osgi.framework
Export-Package: org.jvcode.osgi.provider
```

Das Attribut "Bundle-Activator" definiert die Klasse, welche vom OSGi-Framework beim Aktivieren eines Bundles aufgerufen wird. Das Import-Package-Attribut definiert die Packages als Abängigkeiten, welche das Bundle von anderen Bundles benötigt. Da das "BundleActivator"-Interface vom package "org.osgi.framework" des OSGi-Framework kommt, muss eine Abhängigkeit zu diesem Package definiert werden. Das Attribut "Export-Package" bewirkt genau das Gegenteilige. Hierin wird definiert, welche Packages exportiert werden, und somit anderen Bundles zur Verfügung stehen. Da der HelloWorldService zur Verfügung gestellt werden soll, wird das Package "org.jvcode.osgi.provider" angegeben.

Hinweis
Nach dem Ändern des Manifests will IntelliJ IDEA die Abhängigkeiten synchronisieren. Leider gehen diese dabei verloren, so dass für das Compilieren die Abhängigkeiten zum Apache Felix-Framework und dem HelloWorldConsumer-Bundle manuell bzw. über Quick-Fixes wiederhergestellt werden müssen.

HelloWorldConsumerBundle

Das HelloWorldConsumer-Bundle besteht nur aus einem Bundle-Activator, in dem der vom HelloWorldProvider zur Verfügung gestellte Dienst aufgerufen wird.

```
1  package org.jvcode.osgi.consumer;
2
3  import org.jvcode.osgi.provider.HelloWorldService;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;
6  import org.osgi.framework.ServiceReference;
7
8  public class Activator implements BundleActivator
9  {
10     public void start(BundleContext bundleContext) throws Exception
11     {
12         System.out.println("HelloWorldConsumer bundle is being started...");
13         System.out.println("Getting reference of HelloWorld Service...");
14         ServiceReference reference = bundleContext.getServiceReference(HelloWorldService.class.getName());
15         HelloWorldService helloWorldService = ((HelloWorldService) bundleContext.getService(reference));
16         helloWorldService.helloWorld();
17     }
18
19     public void stop(BundleContext bundleContext) throws Exception
20     {
21         System.out.println("HelloWorldConsumer is being stopped...");
22     }
23 }
```

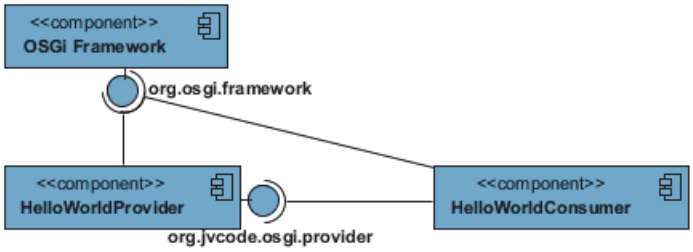
Activator.java

Zuerst wird ein ServiceReference-Objekt unter Angabe des Service-Namens aus dem BundleContext geholt. Da der Service unter dem Namen des Service-Interfaces registriert wurde, kann man hier auch wieder den Klassennamen des Interfaces verwenden. Mit Hilfe der Service-Referenz kann dann der eigentliche Service aus dem BundleContext geholt werden. Zuletzt wird die gewünschte Methode "helloWorld" des Services aufgerufen.
Im obigen Beispiel wurde auf Fehlerbehandlung verzichtet, da eine Fehlersituation unwahrscheinlich ist. In einer Real-Life-Applikation ist diese aber unabdingar, da ein Service mit dem Namen nicht vorhanden sein könnte, das Service-Objekt nicht dem erwarteten Interface entsprechen könnte oder Ähnliches.
Zuletzt muss noch die Manifest-Datei angepasst werden. Zum einen muss noch der BundleActivator aufgenommen werden, zum anderen müssen die benötigten Packages des OSGi-Frameworks und des HelloWorldProvider-Bundles importiert werden.

```
1  Import-Package: org.jvcode.osgi.provider,org.osgi.framework
2  Bundle-Activator: org.jvcode.osgi.consumer.Activator
```

Bundleabhängigkeiten

Das untenstehende Komponenendiagramm zeigt die OSGi-Bundles mit ihren exportierten Packages und die Abhängigkeiten von anderen Bundles zu diesen. Das OSGi-Framework-Bundle exportiert das Package "org.osgi.framework" das von beiden HelloWorld-Bundles für den BundleActivator benötigt wird. Das HelloWorldProvider-Bundle exportiert das Package "org.jvode.osgi.provider", welches das HelloWorldService-Interface und dessen Implementierung enthält. Das HelloWorldConsumer-Bundle exportiert keine Packages. Es benötigt nur die Packages aus dem HelloWordProvider- und OSGi-Bundle.



Demonstrationsclient

Das Beispiel kann mit dem untenstehenden Demonstrationsclient getestet werden. Im ersten Block wird das Apache Felix Framework initialisiert. Der Parameter FRAMEWORK_STORAGE_CLEAN mit dem Wert "onFirstInit" gibt dabei an, dass der Bundle-Cache von Apache Felix bei der Initialisierung geleert wird.

Danach werden die beiden HelloWorld-Bundles installiert. Hierzu muss der Dateiname der Bundles angegeben werden. IntelliJ IDEA erstellt die Bundle-JARs standardmäßig in "out/production". Das Verhalten kann aber in den OSGi-Facet-Einstellungen adaptiert werden.

Zuletzt werden das Framework und die Bundles gestartet und wieder gestoppt.

```java
package org.jvcode.osgi.client;

import org.apache.felix.framework.Felix;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;

import java.util.HashMap;
import java.util.Map;

public class HelloWorldClient
{
    public static void main(String[] args)
    {
        try
        {
            // Initialize Apache Felix Framework
            Map<String, String> configMap = new HashMap<String, String>();
            configMap.put(Constants.FRAMEWORK_STORAGE_CLEAN, "onFirstInit");
            Felix framework = new Felix(configMap);
            framework.init();

            // Install bundles
            BundleContext context = framework.getBundleContext();
            Bundle provider = context.installBundle("file:out/production/HelloWorldProvider.jar");
            Bundle consumer = context.installBundle("file:out/production/HelloWorldConsumer.jar");

            // Start and stop framework and bundles
            framework.start();
            provider.start();
            consumer.start();
            framework.stop();
        }
        catch (Exception exception)
        {
            System.err.println("Error while executing program: " + exception);
            exception.printStackTrace();
            System.exit(0);
        }
    }
}
```

HelloWorldClient.java

Auf der Konsole kann man die Ausführung nachvollziehen. Zuerst wird das Bundle "HelloWorldProvider" gestartet, welches den "HelloWorldService" registriert. Danach wird das "HelloWorldConsumer" Bundle gestartet, welches sich eine Referenz auf den "HelloWorldService" holt. Mit Hilfe dieser Referenz wird der Service aufgerufen, welcher dann denn HelloWorld-String ausgibt. Mit dem Stopp-Befehl für das Apache Felix Framework werden auch die beiden Bundles wieder gestoppt.

```
HelloWorldProvider is being started...
Registering HelloWorldService...
HelloWorldConsumer bundle is being started...
Getting reference of HelloWorld Service...
=== Hello World! ===
HelloWorldConsumer is being stopped...
HelloWorldProvider is being stopped...
```

Ausgabe auf der Konsole

Literaturhinweise

- OSGi-Definition von Wikipedia
  http://de.wikipedia.org/wiki/Osgi
- JSR 338 - JavaSE 8 Release Contents - Modularität für die kommende Java 8 Version
  http://jcp.org/en/jsr/detail?id=337
- Projekt Jigsaw - Draft zur Modularität mit Java 8
  http://openjdk.java.net/projects/jigsaw/doc/draft-java-module-system-requirements-12
- JSR 291 - Dynamic Component Support for Java SE - OSGi als offizieller Bestandteil von Java
  http://jcp.org/en/jsr/detail?id=291
- OSGi in Action, Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage, Manning 2011
  http://www.manning.com/hall/