Plug-in-Programmierung mit Eclipse

Eine kurze Einführung mit Beispielen

Julia Dreier
Hochschule Osnabrück
Version 1.2, 06.10.2010

Inhaltsverzeichnis

Überblick

Eclipse ist eine der bekanntesten Entwicklungsumgebungen unserer Zeit, vor allem für Java. Die Besonderheit besteht darin, dass Eclipse selbst (seit Version 3.0) aus einem Kern besteht, der nur die Plug-ins lädt, die gerade benötigt werden. Dieser Kern ist ein Java-basiertes Framework (Programmiergerüst) namens Equinox, welches die so genannte OSGi-Kernspezifikation implementiert.

Die OSGi-Service-Platform ist ein dynamisches Komponentensystem. Das heißt, dass Plug-ins, die im Zusammenhang mit OSGi auch „Bundles“ genannt werden, von dieser Plattform zur Laufzeit gestartet, aktualisiert und gestoppt werden können, ohne dass Eclipse neu gestartet werden muss. Das OSGi Framework ist die Basiskomponente der OSGi-Service-Platform, die die Laufzeitumgebung für die Bundles zur Verfügung stellt. Im Folgenden werden die Begriffe „Plug-in“ und „Bundle“ synonym verwendet.

Was auf den ersten Blick kompliziert klingt, ist eigentlich ganz einfach: Wir programmieren ein Eclipse-Plug-in im Grunde genommen nicht anders als ein „normales“ Programm. Es gibt keine besonderen Bibliotheken oder Konstrukte, die wir nicht schon aus der allgemeinen Java-Programmierung kennen. Es gibt bloß ein paar Besonderheiten, auf die im Laufe dieses Tutorials noch näher eingegangen wird. Wir könnten im Prinzip jedes Java-Programm, das wir bereits erstellt haben, als ein Eclipse-Plug-in laufen lassen. Ob dies Sinn ergibt, ist allerdings eine andere Frage.

1 Installation der benötigten Software

1.1 Installation des Java Development Kit (JDK)

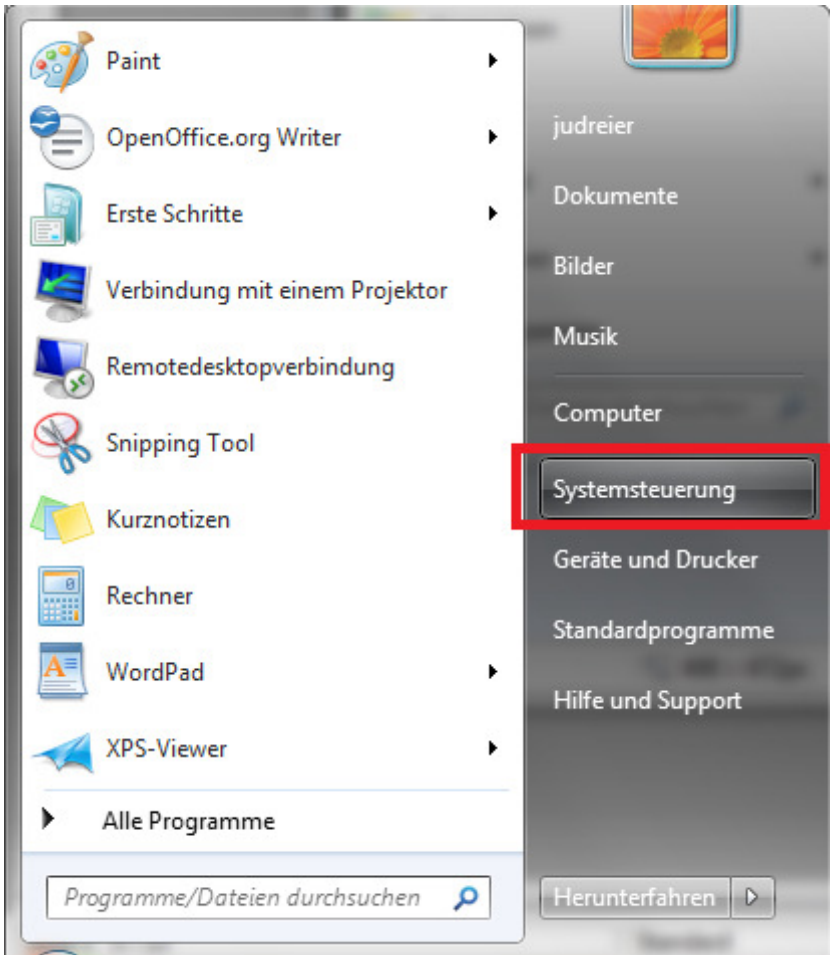
Zuerst wird das aktuelle JDK benötigt, um Java-Quellcode in eine ausführbare Datei kompilieren zu können.

Unter dem Link http://www.oracle.com/technetwork/java/javase/downloads/index.html kann das aktuelle SDK heruntergeladen werden:

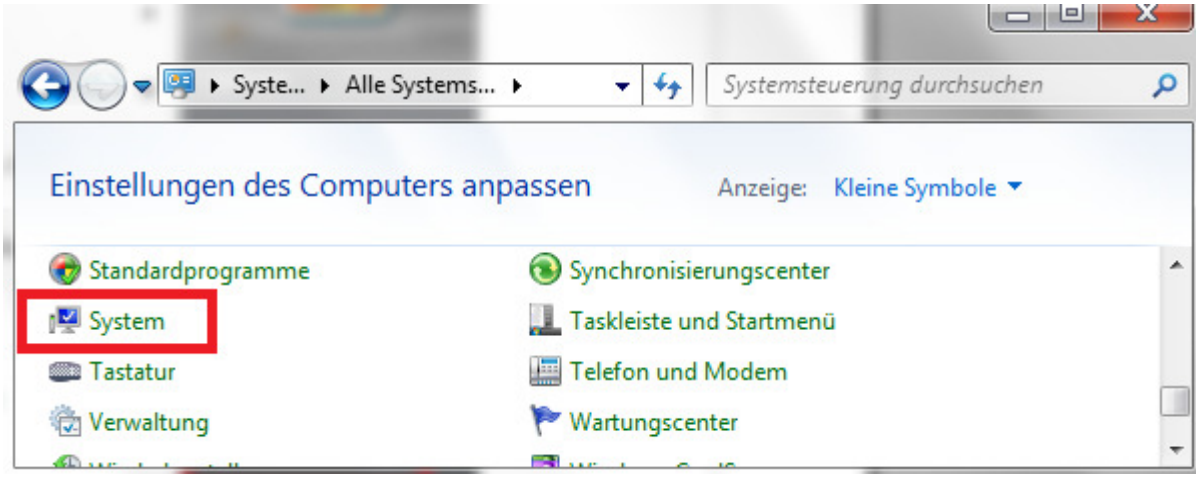
Die heruntergeladene Datei muss dann auf dem jeweiligen System installiert werden. Das Zielverzeichnis spielt dabei keine Rolle.

Damit andere Programme, wie zum Beispiel Eclipse, den Java-Compiler und -Interpreter finden, müssen noch zwei Umgebungsvariablen gesetzt werden. Unter Windows 7 kann dies unter anderem folgendermaßen geschehen:

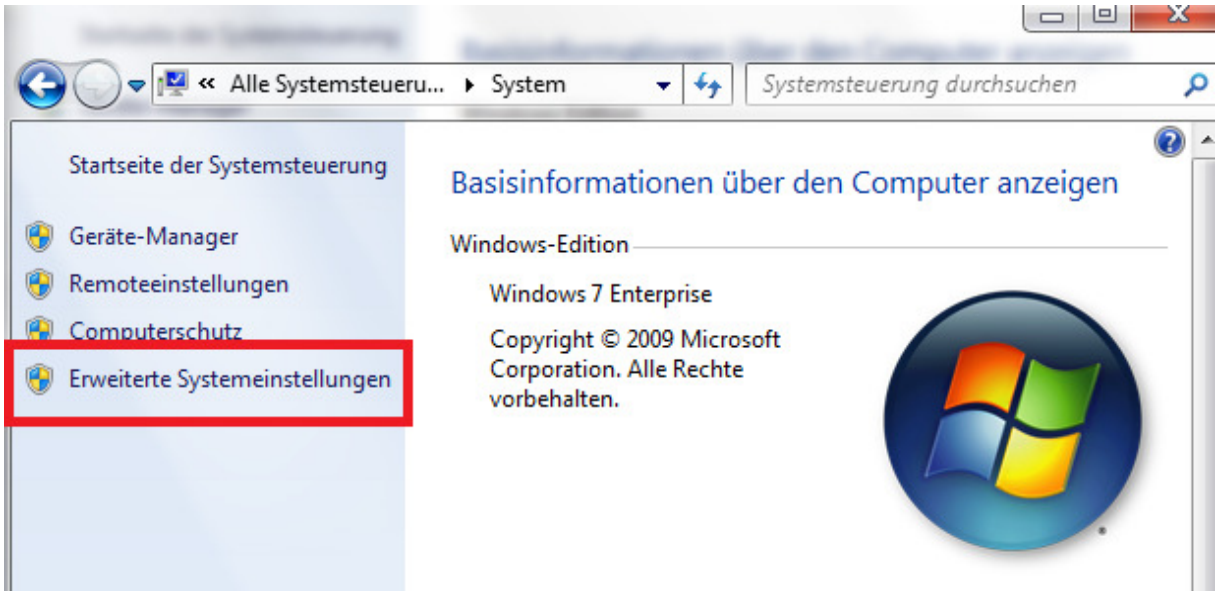
Über das Start-Menü die **Systemsteuerung** aufrufen:



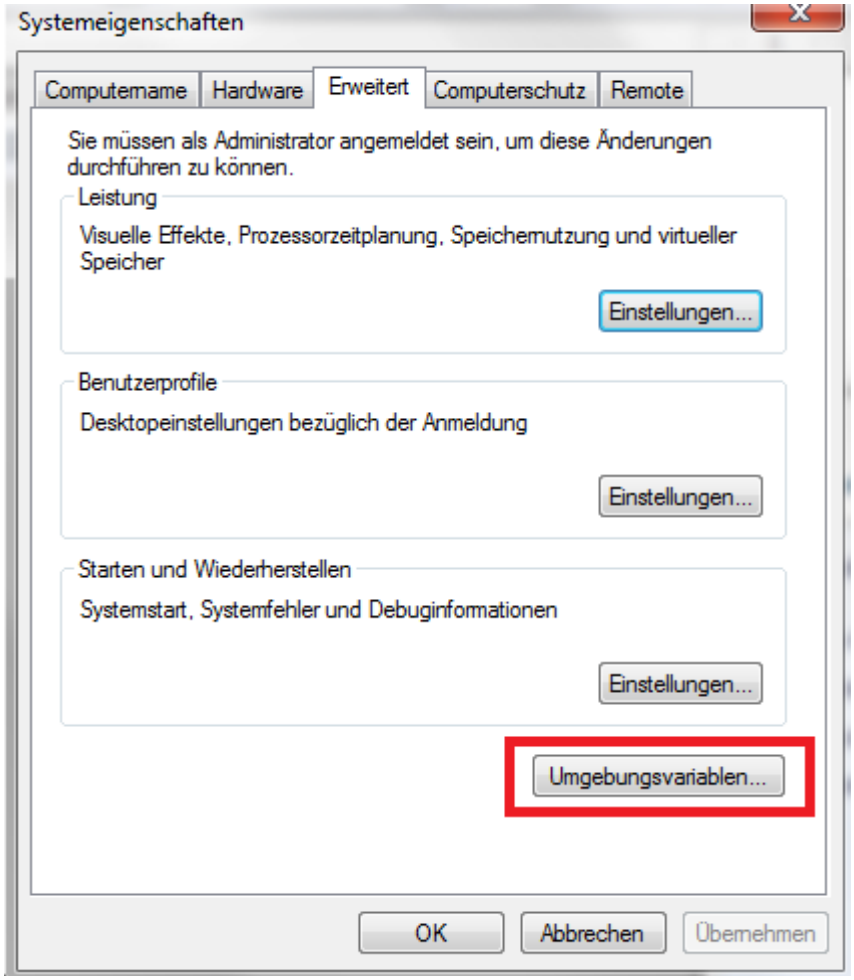
Auswählen des Elements **System**:



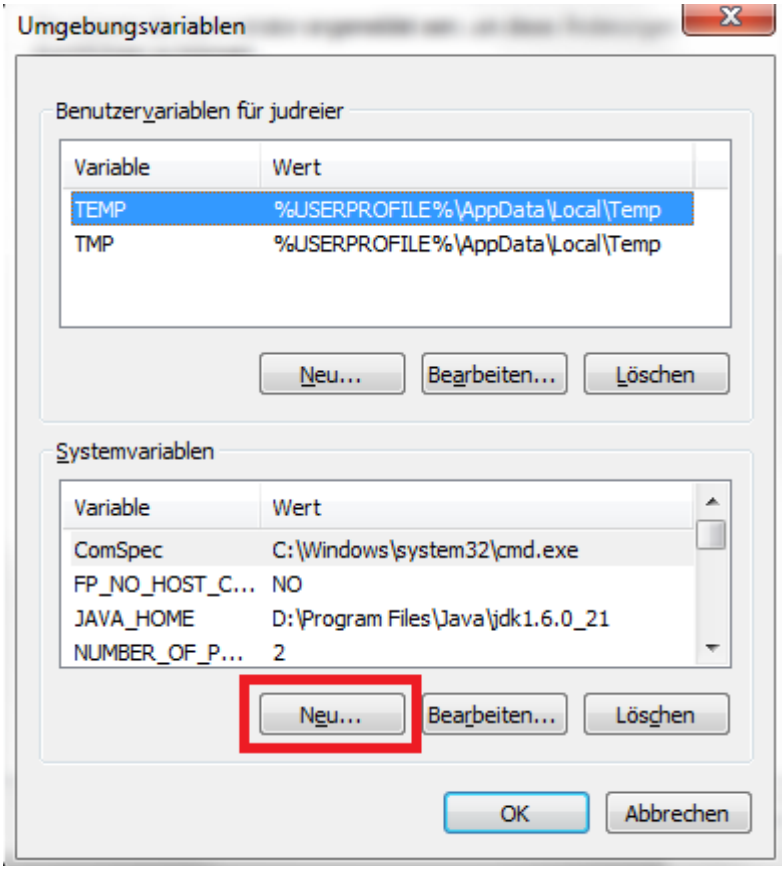
Auswählen des Elements **Erweiterte Systemeinstellungen**:



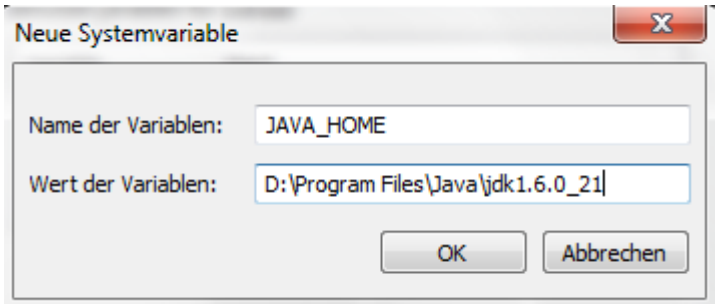
Auswählen des Buttons **Umgebungsvariablen**:



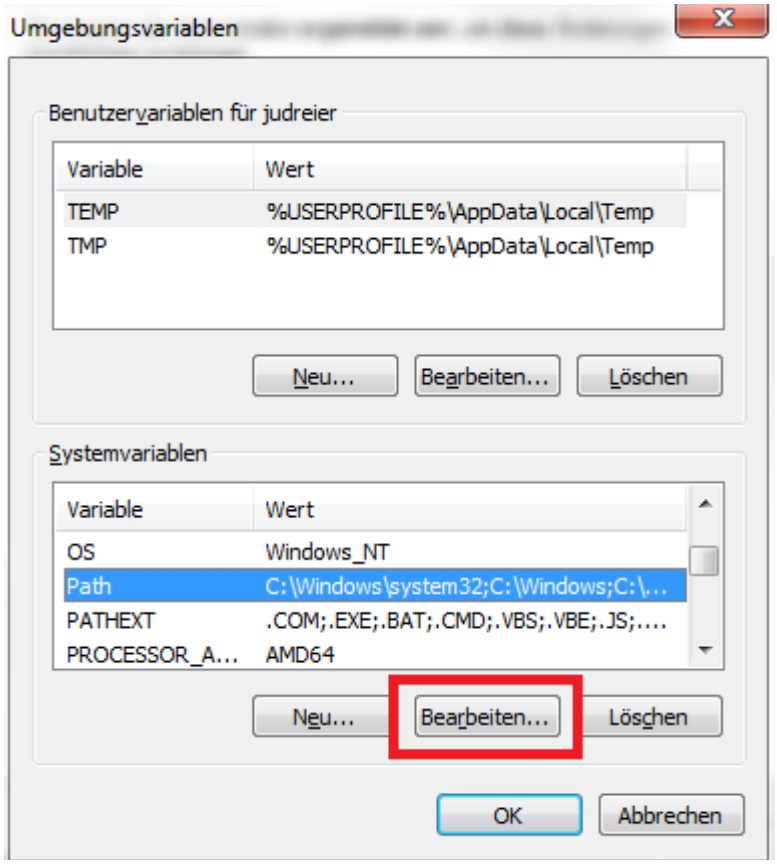
Neue Systemvariable hinzufügen:



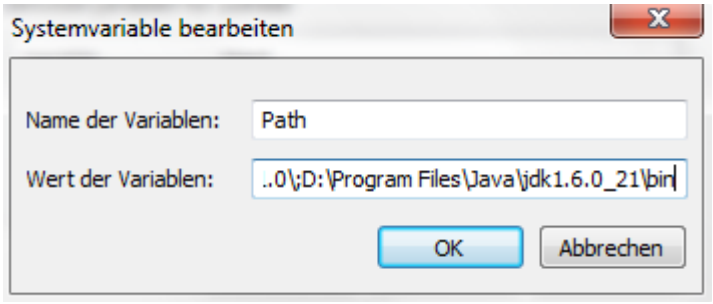
Eingeben des Variablennamens „JAVA_HOME“, als Variablenwert wird das Java-Installationsverzeichnis eingetragen:



Erweitern der Path-Variable:



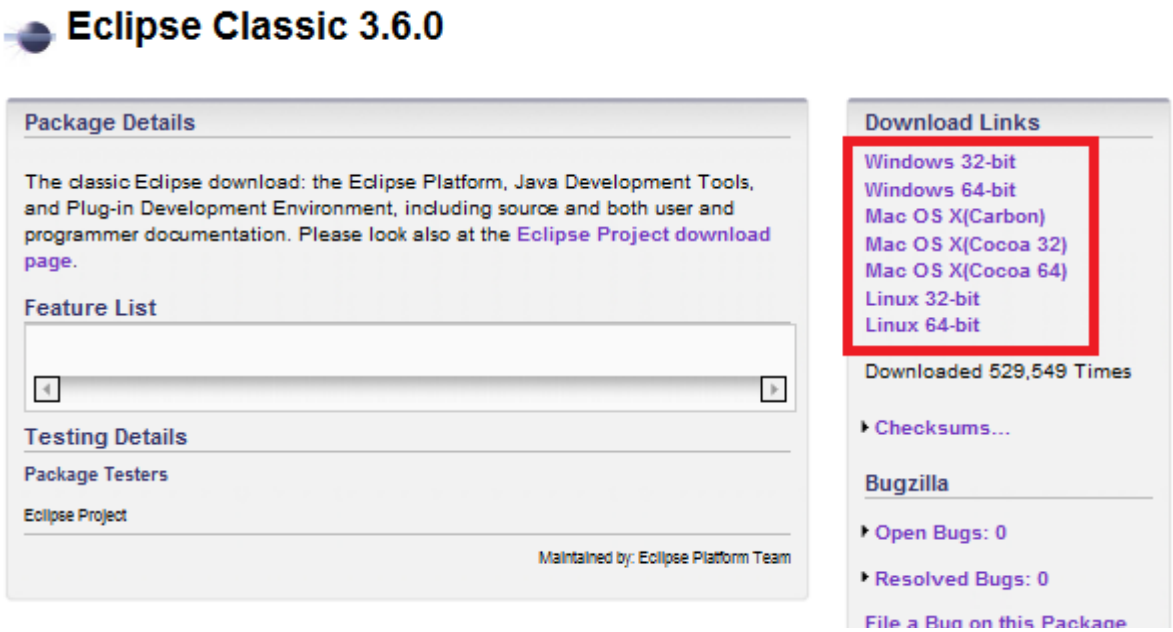
Durch ein Semikolon getrennt wird das bin-Verzeichnis des Java-Installationsverzeichnisses ans Ende des Variablenwertes gehängt:



1.2 Installation von Eclipse

Um Plug-ins mit Eclipse entwickeln zu können, wird „Eclipse Classic“ benötigt. „Eclipse for Java Developers“ reicht in diesem Fall nicht aus.

Die aktuelle Version von Eclipse Classic (Helios, Version 3.6.0) kann unter http://www.eclipse.org/downloads/packages/eclipse-classic-360/heliosr heruntergeladen werden:



Eclipse hat keine typische Installationsroutine. Es muss lediglich der heruntergeladene Ordner entpackt werden. Das Zielverzeichnis spielt dabei keine Rolle.

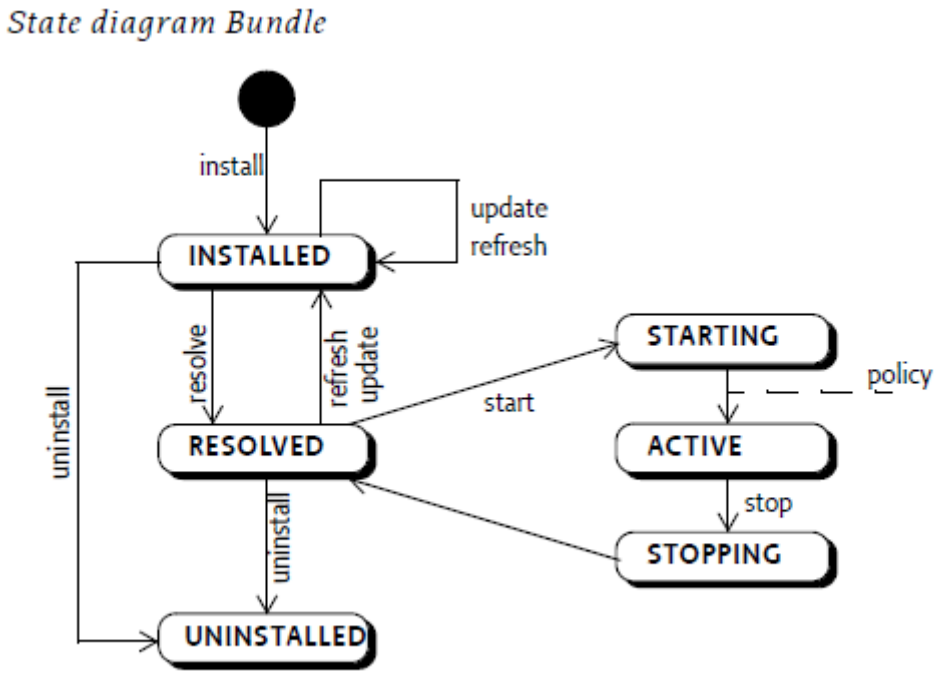
Um Eclipse auszuführen, muss einfach die **eclipse.exe** ausgeführt werden. Möchte man Eclipse wieder deinstallieren, genügt es, den entpackten Ordner wieder zu löschen.

2 Plug-in-Programmierung

Im Folgenden erstellen wir unser erstes Eclipse-Plug-in, das einfach ein paar Ausgaben tätigt. Zunächst werden allerdings ein paar grundlegende Dinge erklärt, die für die Plug-in-Programmierung wissenswert sind.

2.1 Zustände eines Plug-ins

Ein Eclipse-Plug-in kann sechs Zustände annehmen:



Quelle: OSGi Service PlatformCore Specification Release 4, Version 4.2

- ***installed***: das Bundle wurde erfolgreich installiert
- ***resolved***: alle Java-Klassen, die das Bundle benötigt, sind verfügbar; das heißt, dass das Bundle entweder bereit zum Start ist oder gestoppt wurde
- ***starting***: das Bundle wird gestartet

- ***active***: das Bundle wurde gestartet und ist aktiv
- ***stopping***: das Bundle wird gestoppt
- ***uninstalled***: das Bundle ist nicht mehr installiert; es kann in keinen anderen Zustand mehr überführt werden

Den Zustand eines Bundles kann geändert werden, indem bestimmte Aktionen auf diesen Bundles ausgeführt werden. Es gibt zwei Möglichkeiten, den Zustand eines Bundles zu ändern: entweder programmatisch über die Framework-API oder über einen Management Agent wie die Equinox-Konsole.

Wenn ein Bundle seinen Zustand programmatisch im OSGi Framework ändern soll, müssen bestimmte Funktionen (wie z. B. ***start()***) aufgerufen werden. Diese sind über Objekte vom Interface ***Bundle*** zu erreichen. Diese Objekte werden vom Framework für jedes installierte Bundle bereitgestellt.

Wenn der Zustand des eines Bundles „per Hand“ geändert werden soll, kann die Equinox-Konsole verwendet werden. Dort können Kommandos (wie z.B. ***start***) zusammen mit ID oder Namen des Bundles ausgeführt werden, die dann einen Zustandswechsel auslösen.

Um eine gute Performance zu gewährleisten, sollten nur Plug-ins auf den Zustand ***active*** gesetzt werden, wenn sie unmittelbar benötigt werden. Plug-ins, die in nächster Zeit gebraucht werden, sollten auf den Zustand ***resolved*** gesetzt werden. In diesem Zustand sind alle zur Ausführung benötigten Dateien im Framework gecashed und das Plug-in kann bei Bedarf sofort eingesetzt werden.

2.2 Die Activator-Klasse

Ein Bundle sollte immer einen so genannten ***Bundle Activator*** besitzen. Das ist eine besondere Klasse, die das Interface ***osgi.framework.BundleActivator*** implementiert. Diese Klasse entscheidet, was beim Start und Stopp des Bundles passiert. Man kann sich diese Klasse als eine Art Konstruktor bzw. Destruktor für das Plug-in vorstellen.

Es werden die Methoden ***start()*** und ***stop()*** implementiert. Diese werden

automatisch aufgerufen, wenn das Bundle in den Zustand ***starting*** bzw. ***stopping*** übergeht. Hier sollten auch die für das Bundle benötigten Ressourcen reserviert bzw. wieder freigegeben werden.

3 OSGi Services

Ein OSGi Service ist nichts anderes als ein Java-Objekt, das an der so genannten ***Service Registry*** innerhalb des Frameworks angemeldet wird. Dort können diese Services dann von anderen Bundles abgefragt und genutzt werden. Die Registrierung der Objekte erfolgt über das ***Service Interface***, in der Regel der Name des Interfaces bzw. Klassennamen des Java-Objekts, das als Service dienen soll. Um einen Service nutzen zu können, sind folgende Schritte notwendig:

1. **Service registrieren**: Bevor der Service genutzt werden kann, muss er von einem Bundle erzeugt und an der Service Registry angemeldet werden
2. **Service abfragen**: Die an der Service Registry angemeldeten Services können von anderen Bundles über den Klassennamen abgefragt werden
3. **Service nutzen**: Wenn das Bundle einen Service erhalten hat, kann es diesen über das als Service angemeldete Objekt nutzen
4. **Service freigeben**: Wird ein Service nicht mehr genutzt, muss dieser durch das Bundle, das diesen Service beansprucht hat, explizit wieder freigegeben werden
5. **Service deregistrieren**: Wenn ein Service nicht mehr zur Nutzung zur Verfügung gestellt werden soll, dann muss das Bundle, das den Service registriert hat, diesen explizit deregistrieren. Wird das Bundle, das den Service angemeldet hat, allerdings gestoppt, sind auch dessen Services automatisch deregistriert.

Möchte man nun einen Service verwenden, erfolgt der Zugriff mit Hilfe einer so genannten *Service Refrence*, die dazu dient, den konkreten Service von der Service Registry zu erlangen. So können auch das Bundle, das den Service zur Verfügung stellt sowie alle anderen Bundles, die diesen Service nutzen, an der Service Reference abgefragt werden.

4 Plug-ins exportieren

Um Plug-in-Projekte nun auch außerhalb der Eclipse IDE nutzen zu können, müssen die Projekte nun zu „echten“ Bundles gepackt werden. Dazu werden die Projekte mit Hilfe des ***Plug-in-Export-Wizards*** zu einer Jar-Datei gepackt.

Was genau nun alles exportiert werden soll, bleibt uns selbst überlassen. Im Root-Verzeichnis jedes Projekts befindet sich eine Datei ***build.properties***. Auch hierfür stellt Eclipse einen Editor zur Verfügung, der sich mit einem Doppelklick auf die build.properties-Datei öffnet. Unter „Runtime Information“ werden alle Verzeichnisse aufgelistet, die beim Plug-in-Export gepackt werden sollen. Standardmäßig ist dort das aktuelle Verzeichnis (also „.“) eingetragen. Falls weitere Bibliotheken benötigt werden, werden diese dort eingetragen. Unter „Binary Build“ können alle Dateien und Ordner, die in das gepackte Plug-in enthalten soll. Im Standardfall ist dies der META-INF-Ordner. Wenn es gewünscht ist, nicht nur die Klassen und Ressourcen, sondern auch den Quellcode nach dem Export des Plug-ins zur Verfügung zu stellen, müssen diese Dateien im „Source Build“ ausgewählt werden.

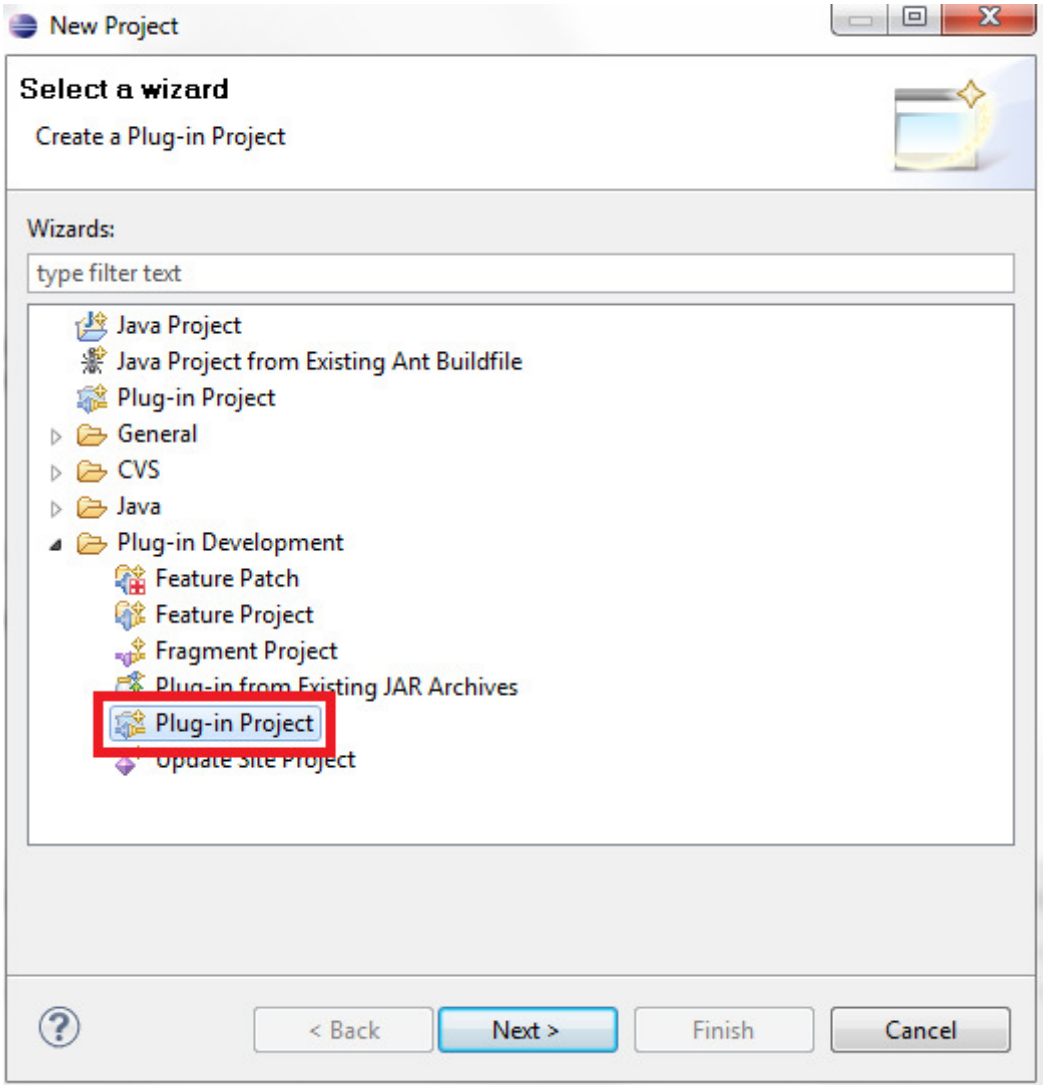
Neben dem konventionellen Weg, ein Plug-in mit Hilfe des Plug-in-Export-Wizards zu exportieren, gibt es auch Szenarien, in denen das Bauen und der Export von Plug-in-Projekten nicht manuell angestoßen werden kann. Dazu gibt es weitere Tools wie der **PDE Headless Build, Ant4Eclipse, Maven** oder **bnd**. Auf diese Möglichkeiten wird hier allerdings nicht weiter eingegangen.

5 Tutorials

Im Folgenden werden einige Beispiele vorgestellt, um zu veranschaulichen, wie Plug-ins innerhalb der Eclipse IDE entwickelt werden. Die Beispiele sind vor allem als Übung gedacht und beinhaltet kein Abfangen von Exceptions etc.

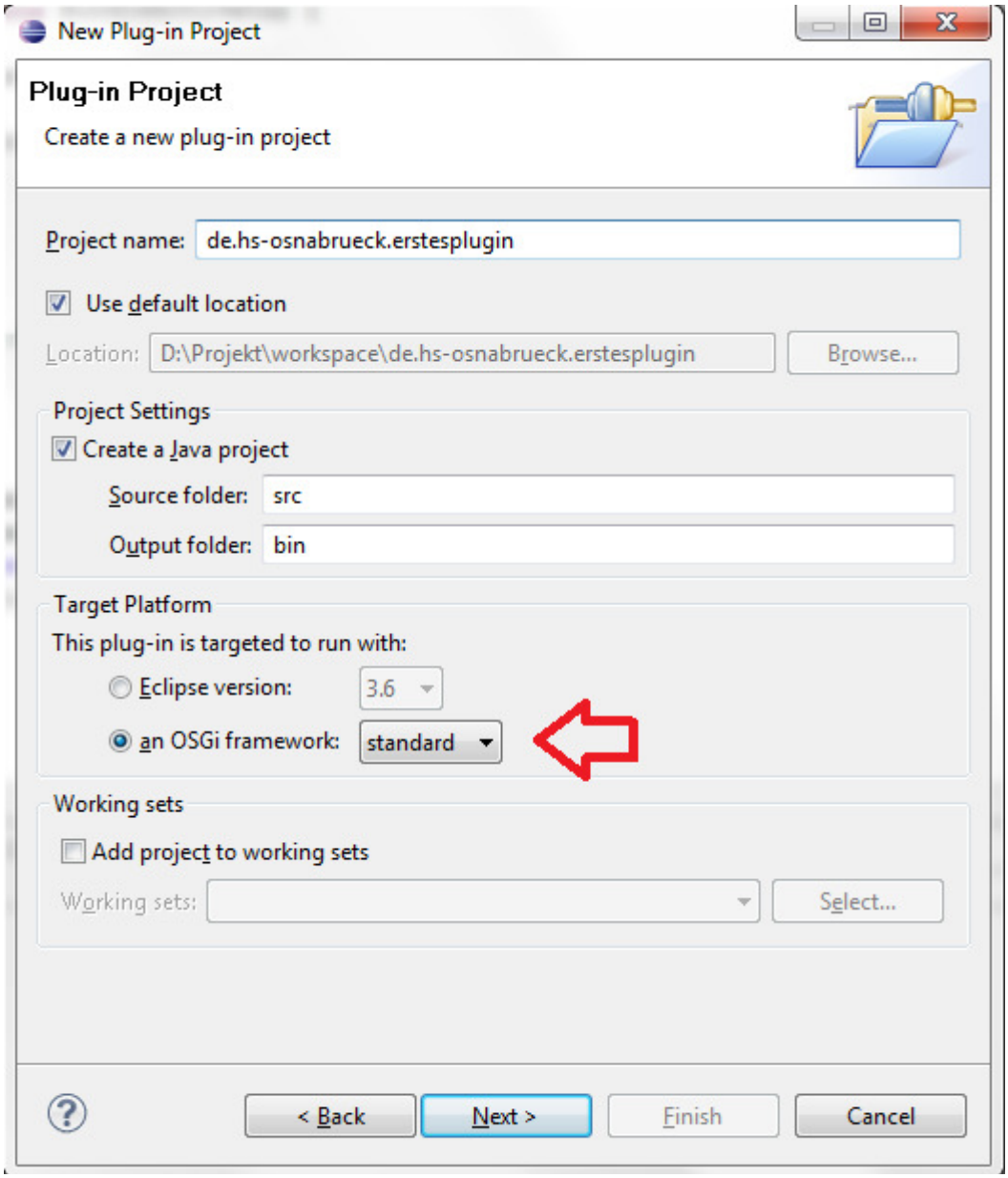
5.1 Ein „Hello World“-Plug-in

Zuerst muss ein neues Projekt angelegt werden. Dafür wird **File** → **New** → **Project...** ausgewählt und **Plug-in-Project** selektiert:



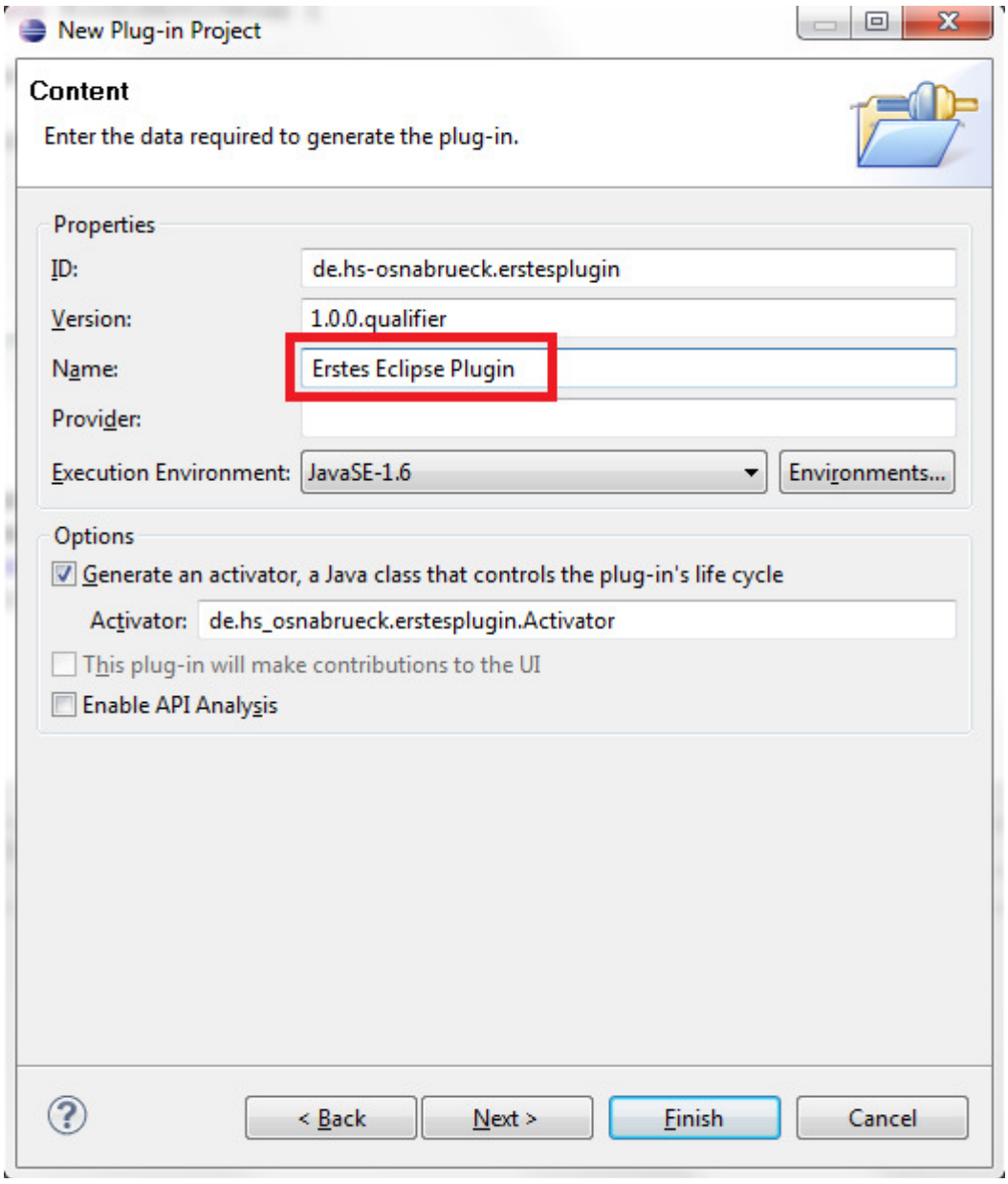
Als nächstes wird dem Projekt ein Name gegeben (hier im Beispiel *de.hs-osnabrueck.erstesplugin*). Bei der Namensgebung des Plug-in-Projekts sollte die Reverse-Domain-Name-Konvention verwendet werden, um Namenskollisionen zu vermeiden. Das heißt, dass der Name mit dem umgekehrten Domain-Namen beginnt. Alle Bundles innerhalb dieses Plug-in-Projekts werden dann mit diesem Projektnamen beginnen. Plug-ins von Eclipse selbst beginnen z. B. immer mit „org.eclipse. ...“.

Für das **OSGi Framework** wird **standard** ausgewählt.

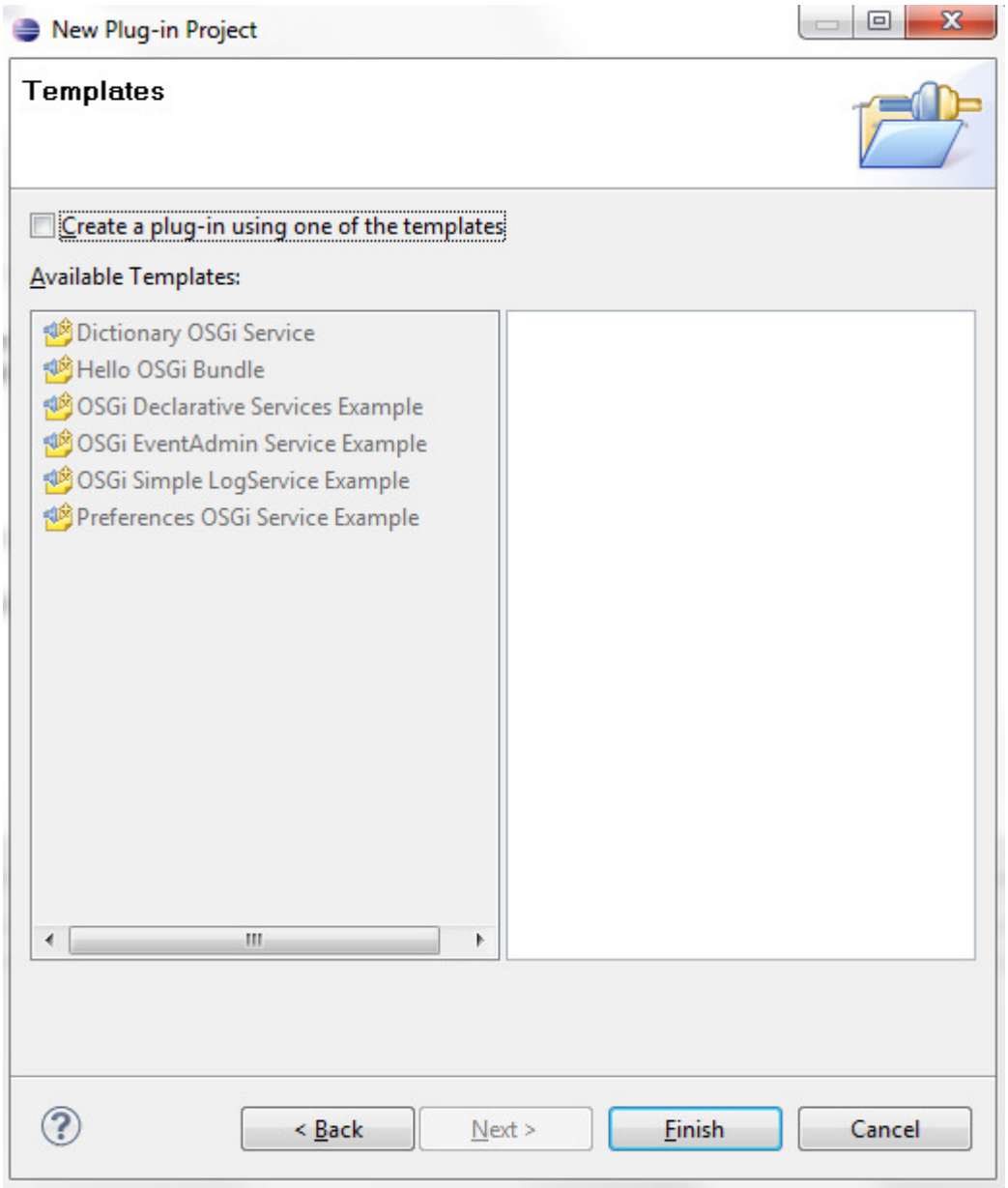


Es kann im Prinzip auch **Equinox** ausgewählt werden. Der Unterschied besteht darin, dass das Plug-in in der Datei ***MANIFEST.MF*** (Datei mit Informationen, die das Framework braucht, um zu wissen, wie das Plug-in ausgeführt werden soll) durch spezifische Equinox-Eigenschaften erweitert wird. Da Equinox nicht das einzige OSGi Framework ist, ist es denkbar, dass das Plug-in später in einem anderen Framework laufen soll. Wählt man **standard** aus, kann das Bundle auch in allen anderen OSGi Release 4-kompatiblen Implementierungen laufen.

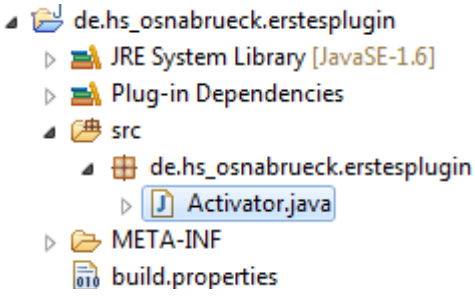
Im nächsten Schritt geben wir unserem Plug-in einen „richtigen“ Namen (hier **Erstes Eclipse Plugin**). Sonst muss nichts verändert werden, die Erstellung einer Activator-Klasse ist bereits vorselektiert.



Im nächsten Schritt können Templates für Plug-ins ausgewählt werden. Das erste Beispiel sollte allerdings erst einmal ohne Templates erstellt werden, um zu lernen, wie die Plug-in-Programmierung funktioniert.



Mit dem Klick auf **Finish** ist das Grundgerüst für das Plug-in erstellt.

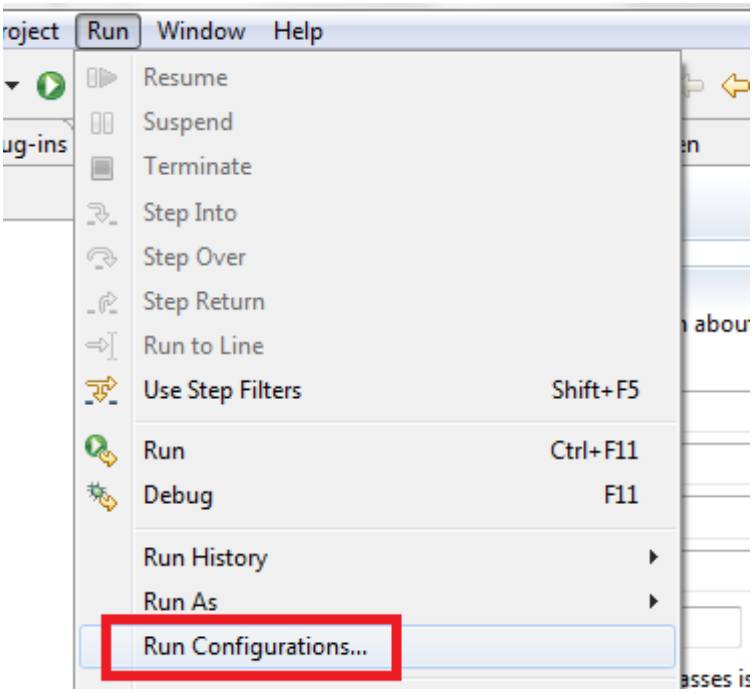


Im Package-Explorer wird die Klasse ***Activator.java*** ausgewählt

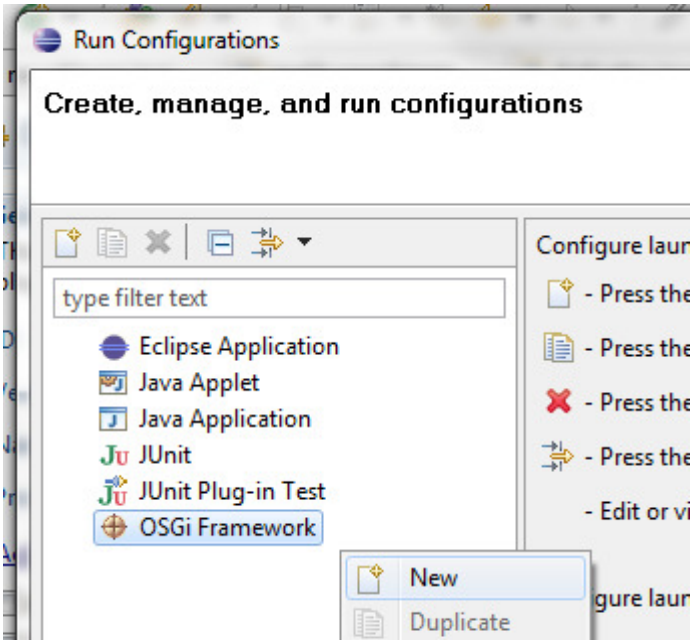
Zu sehen sind jetzt die bereits oben beschriebenen Methoden ***start()*** und ***stop()***. Diese können nun editiert werden. In diesem Beispiel werden zwei Ausgabe-Anweisungen eingefügt, um zu sehen, ob und wann die ***start()***- bzw. ***stop()***-Methode aufgerufen wird.

```
package de.hs_osnabrueck.erstesplugin;

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

public class Activator implements BundleActivator {

	private static BundleContext context;


	static BundleContext getContext() {
		return context;
	}



	public void start(BundleContext bundleContext) throws Exception {
		Activator.context = bundleContext;
		System.out.print("Hello World!");
	}



	public void stop(BundleContext bundleContext) throws Exception {
		Activator.context = null;
		System.out.print("Good Bye World!");
	}


}
```

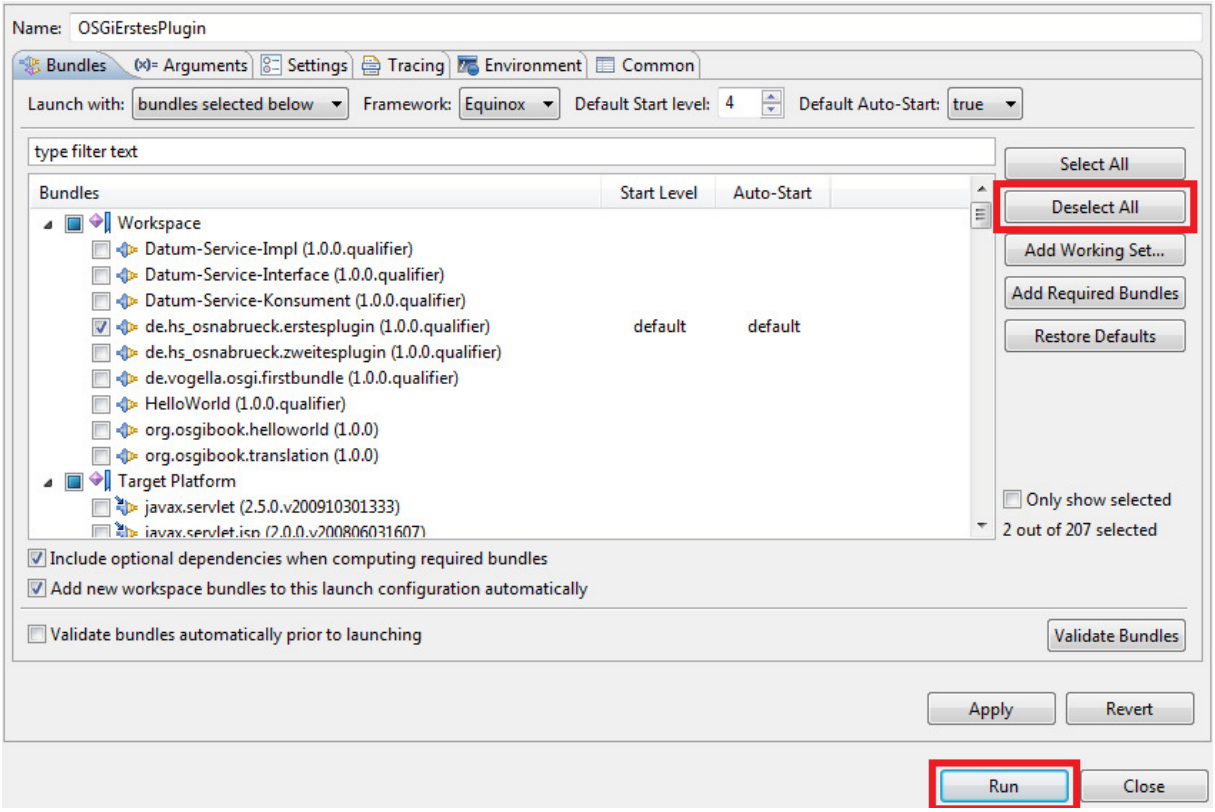
Nun soll das Plug-in zum Laufen gebracht werden. Dazu gehen wir auf **Run** → **Run Configurations**:



Nun kann eine neue OSGi Framework-Konfiguration angelegt werden. Dazu wird **OSGi Framework** selektiert, auf die rechte Maustaste geklickt und **New** gewählt:

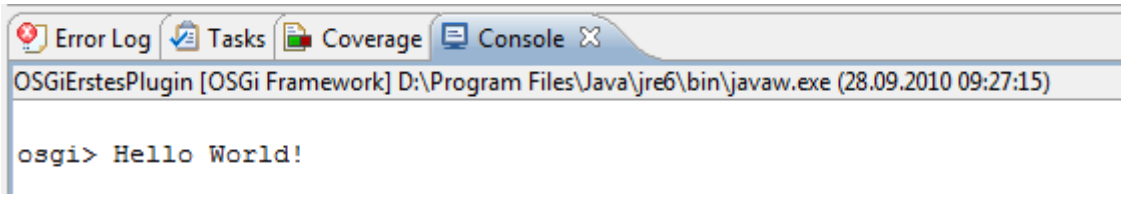


Nun kann eingestellt werden, welche Bundles zur Laufzeit gestartet werden sollen. Dabei stehen eigene Plug-ins sowie die Plug-ins der Eclipse Platform zur Verfügung. Die eigenen Plug-ins sind unter **Workspace** zu finden, bereits mitgelieferte Eclipse-Plug-ins stehen unter **Target Platform**:



Wir nennen die Konfiguration „OSGiErstesPlugin“ und klicken auf „Deselect all“. Nun wird unter **Workspace** nur unser eigenes Plug-in *de.hs_osnabrueck.erstesplugin* und unter **Target Platform** *org.eclipse.osgi* ausgewählt. Danach klicken wir auf „Run“.

Jetzt erscheint auf der Konsole unsere hinzugefügte Ausgabe „Hello World!“:



Für die Administration des OSGi Frameworks wird die Equinox-Konsole verwendet. So kann das Framework von außen gesteuert werden. Zu erkennen ist die Equinox-Konsole am Prompt **osgi>**.

Das Kommando ***ss*** zeigt eine kurze Version der Statusanzeige an. Dadurch wissen wir genau, welche Bundles gerade beim Framework registriert sind und in welchem Zustand sie sich befinden.

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.6.0.v20100517
1       ACTIVE      de.hs_osnabrueck.erstesplugin_1.0.0.qualifier
```

Durch das Kommando ***stop*** [ID des zu stoppenden Bundles] kann ein bestimmtes Bundle angehalten werden. Dieses Bundle geht dann in den Zustand ***resolved*** über.

```
osgi> stop 1
Good Bye World!
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.6.0.v20100517
1       RESOLVED    de.hs_osnabrueck.erstesplugin_1.0.0.qualifier
```

Um das Bundle wieder zu starten, wird das Kommando ***start***[ID des zu startenden Bundles]“ eingegeben.

```
osgi> start 1
Hello World!
```

Soll die OSGi-Sitzung beendet werden, wird das Kommando ***close*** verwendet. Dadurch wird das OSGi Framework heruntergefahren und die virtuelle Maschine beendet.

```
osgi> close

Good Bye World!
```

Weitere Befehle sind in Kapitel 6 **Wichtige OSGi-Kommandos** zu finden.

5.2 Ein „Hello World“-Plug-in mit zwei Klassen

Wie bereits erwähnt, können beliebige Java-Programme auch als OSGi-Plug-in laufen. In diesem weiteren Beispiel wollen wir eine kleine Anwendung schreiben, die mit Hilfe der Swing-Bibliothek ein Ausgabe-Fenster öffnet.

Hier gehen wir genauso vor wie im ersten Beispiel: Es wird ein neues Plug-in-Projekt namens *de.hs_osnabrueck.zweitesplugin* mit folgender Activator-Klasse angelegt:

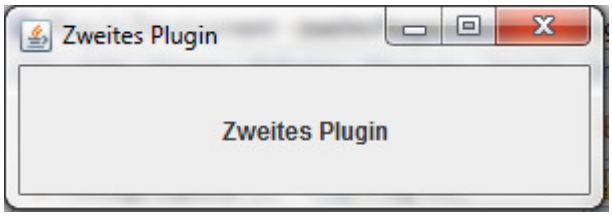
```
package de.hs_osnabrueck.zweitesplugin;

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

public class Activator implements BundleActivator {

	private static BundleContext context;
	private Fenster fenster;


	static BundleContext getContext() {
		return context;
	}



	public void start(BundleContext bundleContext) throws Exception {
		Activator.context = bundleContext;
		System.out.print("Hello World!");
		fenster = new Fenster();
	}



	public void stop(BundleContext bundleContext) throws Exception {
		System.out.print("Good Bye World!");
		Activator.context = null;
	}


}
```

Im src-Ordner des Plug-ins fügen wir dem Package *de.hs_osnabrueck.zweitesplugin* eine weitere Klasse namens *Fenster* hinzu. Dort fügen wir folgenden Code ein:

```
package de.hs_osnabrueck.zweitesplugin;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class Fenster extends JFrame{

	public Fenster() {
		super("Zweites Plugin");

		JLabel label = new JLabel("Zweites Plugin");
		label.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
		getContentPane().add(label);

		setSize(300, 100);

		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		setVisible(true);
	}
}
```

Dann müssen analog zum ersten Beispiel die **Run Configurations** eingestellt werden. Dazu wählen wir dieses Mal nur unser Plug-in *de.hs_osnabrueck.zweitesplugin* aus dem Workspace und wieder *org.eclipse.osgi* aus der Target Platform aus.

Mit dem Klick auf „Run“ startet unsere Anwendung. In der Konsole wird die Ausgabe aus der ***start()***-Methode getätigt und unser neues Fenster wird geöffnet.



5.3 „Hello World“ mit OSGi Services

Wir werden nun zwei Bundles erstellen: zuerst entwickeln wir das Bundle *Producer*, das den Service zur Verfügung stellen wird und dann das zweite Bundle *Consumer*, das den Service des Producers nutzen wird. Dazu werden wir zwei Plug-in-Projekte erstellen: *de.hs_osnabrueck.producer* und *de.hs_osnabrueck.consumer.* Die Einstellungen sind aus dem ersten Tutorial zu entnehmen.

Auch bei kleineren Programmen sollten die Prinzipien der objektorientierten Programmierung stets beachtet werden. Eine der wichtigsten Regeln besagt, dass auf eine Schnittstelle und nicht auf eine Implementierung programmiert werden soll. Das heißt, dass wir zuerst ein Interface mit abstrakten Methoden entwerfen, das den *Hello-World-Service* für uns definiert:

```
package de.hs_osnabrueck.producer;

public interface HelloWorld {
	void sayHello();
	void sayGoodBye();
}
```

Nun erstellen wir ein neues Package innerhalb des Producer-Projekts namens *de.hs_osnabrueck.producer.impl*. Die Activator-Klasse wird in dieses Package verschoben, sodass im Package *de.hs_osnabrueck.producer* nur das Interface übrig bleibt. Innerhalb des neuen Packages *de.hs_osnabrueck.producer.impl* erstellen wir eine Klasse namens *HelloWorldImpl,* die die abstrakten Methoden des *HelloWorld*-Interfaces implementiert. Wir fügen folgenden Code ein:

```
package de.hs_osnabrueck.producer.impl;

import de.hs_osnabrueck.producer.HelloWorld;


public class HelloWorldImpl implements HelloWorld{

	@Override
	public void sayHello() {
		System.out.println("Hello World!");
	}

	@Override
	public void sayGoodBye() {
		System.out.println("Good Bye World!");
	}
}
```

Die Activator-Klasse enthält folgenden Code:

```
package de.hs_osnabrueck.producer.impl;

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

import de.hs_osnabrueck.producer.HelloWorld;

public class Activator implements BundleActivator {

	private static BundleContext context;


	static BundleContext getContext() {
		return context;
	}



	public void start(BundleContext bundleContext) throws Exception {
		Activator.context = bundleContext;

		System.out.println("Producer: Service registrieren...");
		context.registerService(HelloWorld.class.getName(),
			new HelloWorldImpl(), null);
	}



	public void stop(BundleContext bundleContext) throws Exception {
		// Deregistrierung des Services erfolgt automatisch
	}


}
```

Die hier blau markierte Codezeile ist für Bundles, die Services zur Verfügung stellen wollen, unumgänglich. Der ***start()***- und auch der ***stop()***-Methode der Activator-Klassen wird als Parameter der ***BundleContext*** übergeben. Dieses Interface wird vom OSGi Framework implementiert und dazu genutzt, um aus einem Bundle heraus auf das Framework zugreifen zu können. In diesem Fall wird die Methode ***registerService()*** verwendet, um den Service unter einer Klasse bzw. Interface (hier *HelloWorld*) und einem Service-Objekt (hier ein Objekt vom Typ *HelloWorldImpl*) zu registrieren. Der dritte Parameter (hier *null*) steht für eine Liste von Properties, die zur weiteren Beschreibung des Service dient. Außerdem können die Properties verwendet werden, um beim Abfragen einer Service Reference von der Service Registry bestimmte Services zu filtern. In diesem Beispiel werden wir die Properties allerdings nicht näher betrachten.

Nun muss die Activator-Klasse des Packages *de.hs-osnabrueck.consumer* noch angepasst werden:

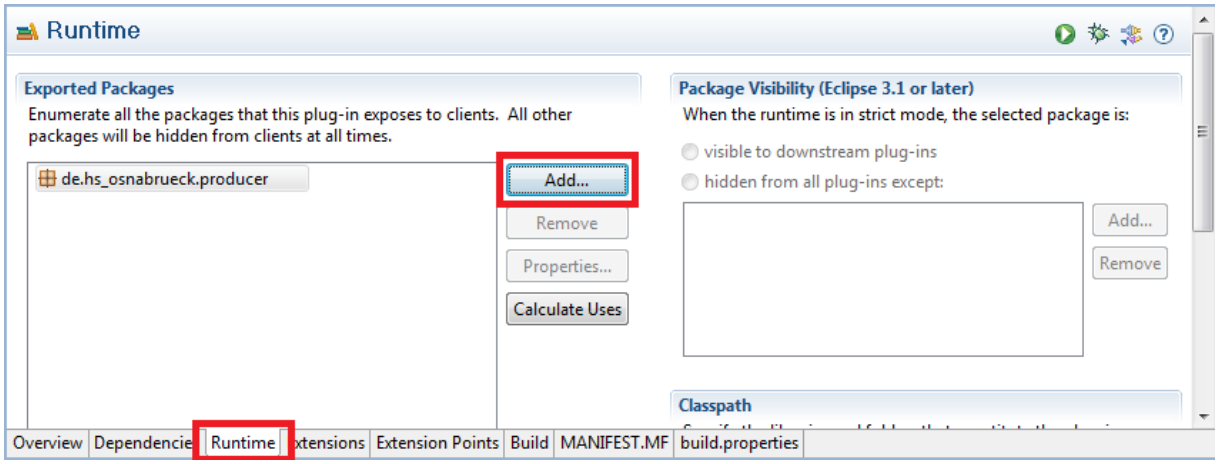
```
package de.hs_osnabrueck.consumer;

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


import de.hs_osnabrueck.producer.HelloWorld;

public class Activator implements BundleActivator {

	private static BundleContext context;
	private HelloWorld service;


	static BundleContext getContext() {
		return context;
	}



	public void start(BundleContext bundleContext) throws Exception {
		Activator.context = bundleContext;

		System.out.println("Consumer: Service anfordern...");
		ServiceReference reference =
			context.getServiceReference(HelloWorld.class.getName());
		service = (HelloWorld) context.getService(reference);
		service.sayHello();

	}



	public void stop(BundleContext bundleContext) throws Exception {
		service.sayGoodBye();
		Activator.context = null;

	}


}
```

Hier holt sich der Consumer mit Hilfe der Methode ***getServiceReference()*** die ***Service Reference*** der Klasse *HelloWorld*. Diese Referenz casten wir und weisen sie einer lokalen Variable der Activator-Klasse zu (hier *service*). So können wir dieses Objekt jetzt wie gewohnt verwenden.

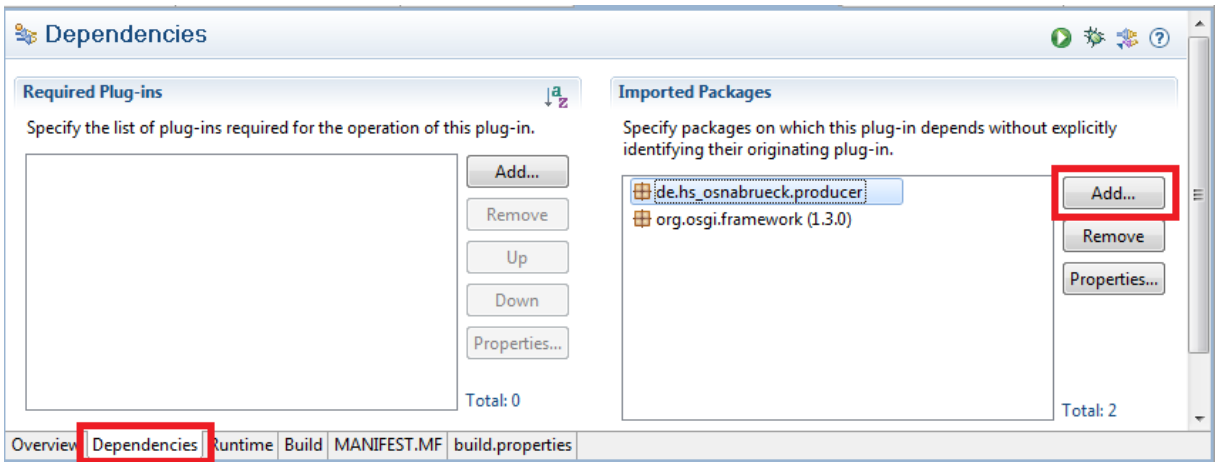
Wie bereits erwähnt müssen Packages, die auch in anderen Projekten verwendet werden sollen, explizit im- und exportiert werden, damit die Bundles miteinander interagieren können. Dies geschieht über die Datei ***MANIFEST.MF*** im Ordner ***META-***

INF des jeweiligen Plug-in-Projekts. Für diese Datei stellt Eclipse einen eigenen Manifest-Editor zur Verfügung, der sich mit einem Doppelklick auf eine Manifest-Datei öffnet.

Damit der Consumer auf das Interface *HelloWorld* zugreifen kann, muss das Package, in dem dieses Interface liegt, exportiert werden. Dazu wird die Manifest-Datei des Producer-Projekts geöffnet und der Reiter **Runtime** ausgewählt. Dann wird über **Add** das benötigte Package *de.hs_osnabrueck.producer* hinzugefügt.

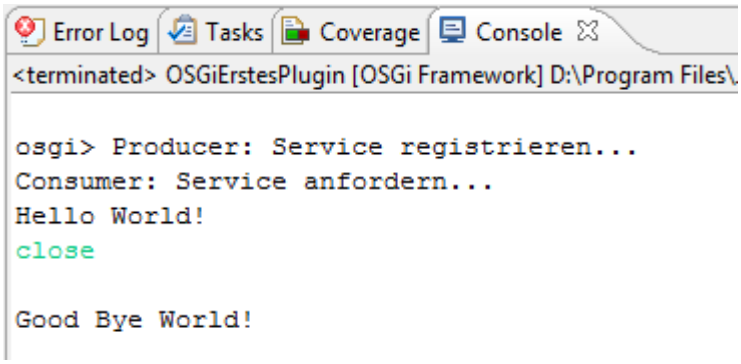


Umgekehrt muss nun der Consumer das benötigte Package *de.hs_osnabrueck.producer* importieren. Dies geschieht über die Manifest-Datei des Consumer-Projekts im Reiter **Dependencies**.



Nun können wir unsere Bundles ausführen. Dazu werden bei den **Run Configurations** im Workspace *de.hs_osnabrueck.producer* sowie *de.hs_osnabrueck.consumer* und bei „Target Platform wieder *org.eclipse.osgi* ausgewählt.

Jetzt können wir sehen, wie der Producer den Service registriert, der Consumer den Service anfordert und dann die Methode *sayHello()* ausgeführt wird.



Es gibt allerdings ein großes Problem, was bei dieser Art von Service-Nutzung auftritt. Würden wir jetzt beide Bundles stoppen und das Consumer-Bundle zuerst wieder starten, würde es einen Fehler geben, da der angeforderte Service (noch) gar nicht existiert. Um dieses Problem und somit Abhängigkeiten zwischen den Bundles zu vermeiden, werden wir dieses Beispiel um einen so genannten ***Service Tracker*** erweitern.

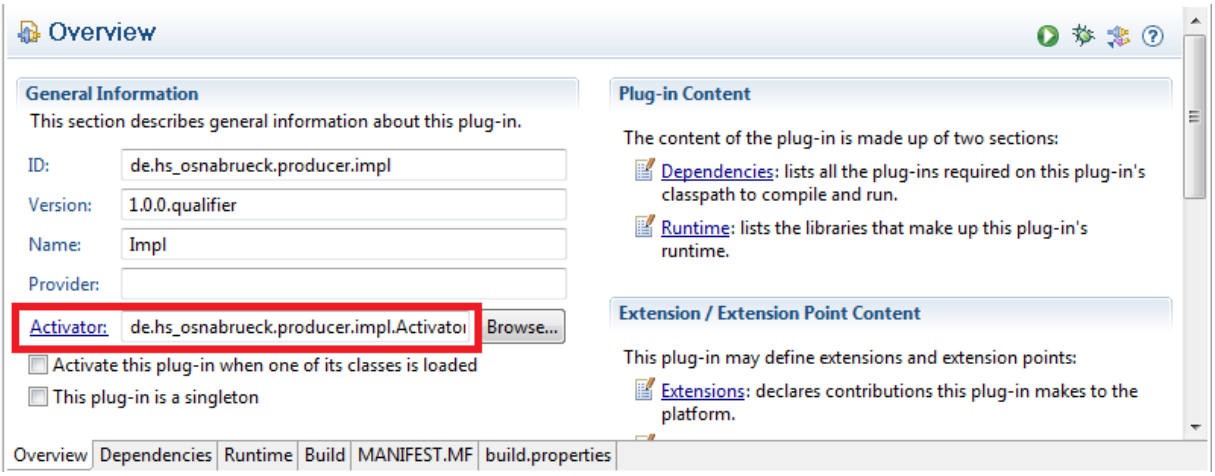
5.4 „Hello World“ mit OSGi Services und ServiceTracker

In diesem Beispiel werden wir das vorige Beispiel als Grundlage nutzen. Wir werden im Folgenden zwei Dinge modifizieren: unser Producer-Bundle in zwei Bundles aufteilen und einen ***Service Tracker*** implementieren.

Wenn wir das Producer-Bundle so aufteilen, dass wir je ein Bundle für die Schnittstelle und die Implementierung erhalten, kann das Consumer-Bundle, das die Schnittstellenklassen für unseren *Hello-World-Service* benötigt, gestartet werden, ohne dass die Implementierung im System installiert ist.

Den Service Tracker wird verwendet, um auf die dynamische Registrierung und Deregistrierung unseres *Hello-World-Services* zu reagieren.

Zuerst legen wir ein neues Plug-in-Projekt *de.hs_osnabrueck.producer.impl* ohne Activator-Klasse an. Dann wird das Package *de.hs_osnabrueck.producer.impl* aus dem Projekt *de.hs_osnabrueck.producer* in das neue Projekt verschoben. Damit das neue Bundle auf das Interface *HelloWorld* zugreifen kann, muss das Package *de.hs_osnabrueck.producer* importiert werden. Da wir auch den Bundle-Activator verschoben haben, muss dieser als Activator im neuen Bundle festgelegt werden. Dazu wird in der Manifest-Datei des Packages *de.hs_osnabrueck.producer.impl* unter dem Reiter **Overview** die verschobene Activator-Klasse eingetragen:



Analog dazu muss natürlich im Producer-Projekt der Eintrag der Activator-Klasse gelöscht werden. Außerdem muss das Package *de.hs_osnabrueck.producer* auch in das neue Projekt importiert werden (wieder über den Reiter **Dependencies** in der Manifest-Datei). Damit wir einen ***Service Tracker*** nutzen können, muss das Package *org.osgi.util.tracker* in das Consumer-Projekt importiert werden.

Zur Kontrolle hier alle Manifest-Dateien (in Textform zu finden auf dem Reiter „MANIFEST.MF“ in den einzelnen Manifest-Dateien) auf einen Blick:

- MANIFEST.MF von *de.hs_osnabrueck.producer*:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Producer
Bundle-SymbolicName: de.hs_osnabrueck.producer
Bundle-Version: 1.0.0.qualifier
Import-Package: org.osgi.framework;version="1.3.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: de.hs_osnabrueck.producer
```

- MANIFEST.MF von *de.hs_osnabrueck.producer.impl*:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Producer Implementation
Bundle-SymbolicName: de.hs_osnabrueck.producer.impl
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: de.hs_osnabrueck.producer.impl.Activator
Import-Package: de.hs_osnabrueck.producer,
 org.osgi.framework;version="1.3.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

- MANIFEST.MF von *de.hs_osnabrueck.consumer*:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Consumer
Bundle-SymbolicName: de.hs_osnabrueck.consumer
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: de.hs_osnabrueck.consumer.Activator
Import-Package: de.hs_osnabrueck.producer,
 org.osgi.framework;version="1.3.0",
 org.osgi.util.tracker
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Der Code der beiden Producer-Projekte muss nicht verändert werden. Dafür wird dem Consumer der benötigte ***Service Tracker*** als innere Klasse hinzugefügt:

```
package de.hs_osnabrueck.consumer;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.util.tracker.ServiceTracker;


import de.hs_osnabrueck.producer.HelloWorld;

public class Activator implements BundleActivator {

	private static BundleContext context;
	private ServiceTracker helloWorldServiceTracker;


	static BundleContext getContext() {
		return context;
	}



	public void start(BundleContext bundleContext) throws Exception {
		context = bundleContext;

		System.out.println("Consumer: Service anfordern...");
		helloWorldServiceTracker = new HelloWorldServiceTracker(context);
		helloWorldServiceTracker.open();
	}



	public void stop(BundleContext bundleContext) throws Exception {
		helloWorldServiceTracker.close();
		Activator.context = null;
	}


	class HelloWorldServiceTracker extends ServiceTracker{

		public HelloWorldServiceTracker(BundleContext bundleContext){
			super(bundleContext, HelloWorld.class.getName(), null);
		}

		public Object addingService(ServiceReference reference){
			HelloWorld helloWorld =
				(HelloWorld)context.getService(reference);
			helloWorld.sayHello();
			return helloWorld;
		}

		public void removedService(ServiceReference reference,Object service){
			HelloWorld helloWorld =
				(HelloWorld)context.getService(reference);
			helloWorld.sayGoodBye();
			context.ungetService(reference);
		}
	}
}
```

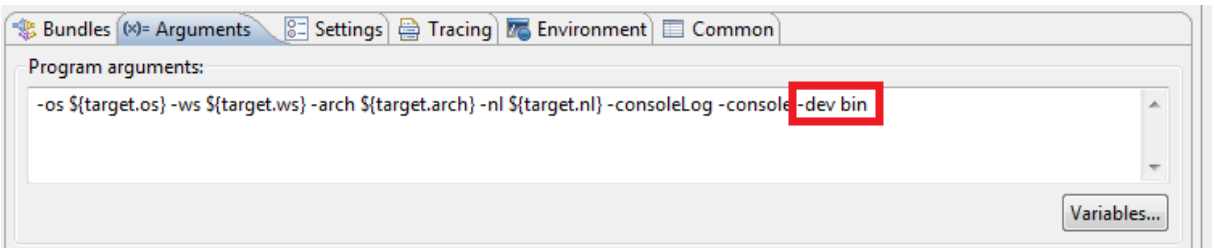
Der Service Tracker kann natürlich auch als eigentständige Klasse implementiert werden. Da die Activator- und die ServiceTracker-Klasse jedoch sehr eng miteinander verbunden sind, ist es sinnvoll, zusammengehörige Klassen hier auch zusammenzufassen.

Über diesen Service Tracker werden wir nun über Änderungen an der Service Registry benachrichtigt. Um mit der Beobachtung der Services zu beginnen, muss der Service Tracker explizit mit der Methode ***open()*** geöffnet werden. Analog dazu muss der Service Tracker auch wieder mit Hilfe der Methode ***close()*** geschlossen werden, wenn er nicht mehr verwendet werden soll.

Wenn ein passender Service in der Service Registry vorhanden ist, ruft der Service Tracker die Methode ***addingService()*** auf. Wenn ein Service aus der Service Registry entfernt wird, wird die Methode ***removedService()*** aufgerufen.

Nun wollen wir unsere Bundles zum Laufen bringen. Dazu werden wieder **Run Configurations** aufgerufen und nur die Bundles *de.hs_osnabrueck.producer* und *de.hs_osnabrueck.consumer* im Workspace ausgewählt (bei Target Platform muss wie immer *org.eclipse.osgi* ausgewählt werden). Des Weiteren muss ein so genanntes ***Program Argument*** eingetragen werden, um zu garantieren, dass die Plug-in-Projekte zur Laufzeit korrekt installiert und ausgeführt werden können. Da die Plug-ins in Eclipse direkt aus den Projekten gestartet werden können, ohne dass sie explizit als Jar-Datei gepackt werden müssen, muss dem Framework mitgeteilt werden, wo sich die Klassen und Ressourcen befinden. Also müssen wir die Ausgabeverzeichnisse des Projekts übergeben. Im Standardfall ist das das bin-Verzeichnis.

Dazu wird der Reiter **Arguments** aufgerufen und das Argument **-dev bin** hinzugefügt.



Nun starten wir unsere Konfiguration:

```
osgi> Consumer: Service anfordern...
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.6.0.v20100517
5       ACTIVE      de.hs_osnabrueck.consumer_1.0.0.qualifier
10      ACTIVE      de.hs_osnabrueck.producer_1.0.0.qualifier
```

Bis jetzt sind wie gewollt nur der Producer und der Consumer gestartet. Nun installieren wir das Projekt *de.hs_osnabrueck.producer.impl* manuell nach. Dazu wird **install file:**/ gefolgt vom Dateipfad des Ordners, in dem das Projekt liegt, in die Konsole eingegeben:

```
osgi> install file:/D:\meine_dateien\Workspace\de.hs_osnabrueck.producer.impl
Bundle id is 11

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.6.0.v20100517
5       ACTIVE      de.hs_osnabrueck.consumer_1.0.0.qualifier
10      ACTIVE      de.hs_osnabrueck.producer_1.0.0.qualifier
11      INSTALLED   de.hs_osnabrueck.producer.impl_1.0.0.qualifier
```

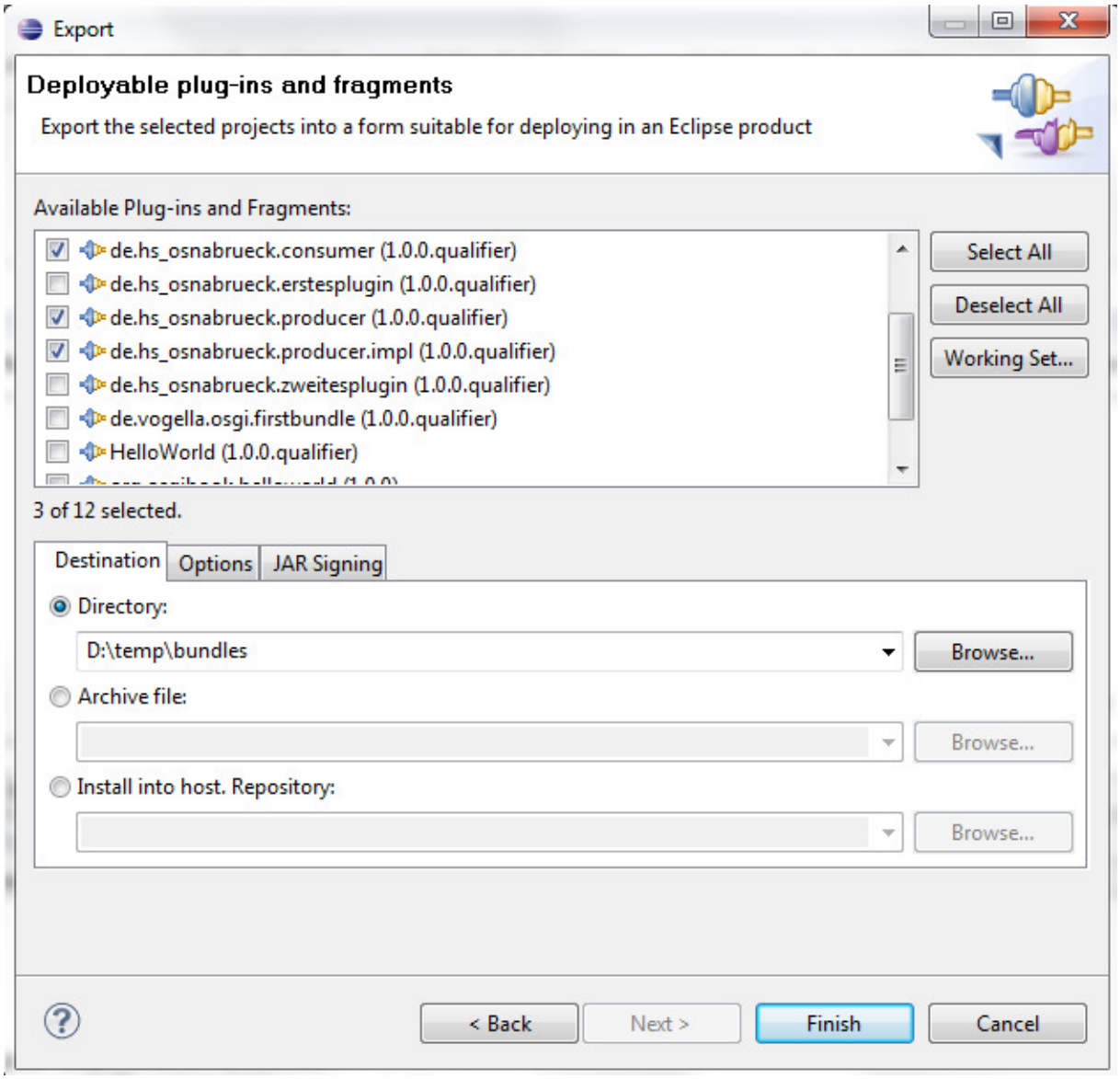
Nun kann das Bundle gestartet und gestoppt werden, der Service Tracker des Consumers wird dies immer registrieren und die Methoden ***addingService()*** bzw. ***removedService()*** aufrufen:

```
osgi> start 11
Producer: Service registrieren...
Hello World!

osgi> stop 11
Good Bye World!

osgi> start 11
Producer: Service registrieren...
Hello World!
```

4.5 „Hello World“ exportieren

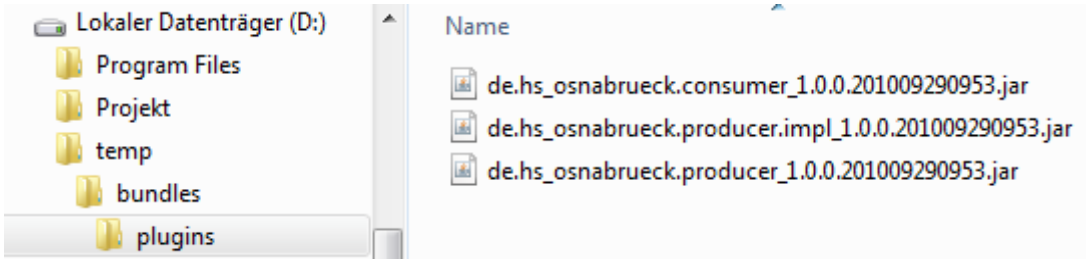
In unserem Fall müssen keine Änderungen an der ***build.properties***-Datei vorgenommen werden. Diese Datei dient der Angabe, wo sich Quellcode-Verzeichnis(se) und Compiler-Ausgabeverzeichnis befinden.

Um das Plug-in zu exportieren, wird der Plug-in-Export-Wizard aufgerufen. Dazu gehen wir auf **File** → **Export** und wählen **Deployable plug-ins and fragments** aus. Dann werden uns alle Plug-in-Projekte angezeigt:



In diesem Tutorial werden wir die Bundles des vorigen Tutorials exportieren. Die Schritte zum Export können allerdings auf beliebige Projekte angewendet werden.

Um die Dateien exportieren zu können, müssen sie lediglich im Wizard ausgewählt, der Zielort bestimmt und auf **Finish** geklickt werden.



Wir sehen, dass unsere Projekte zu je einer Jar-Datei gepackt wurden. Nun kann die Equinox-Konsole ausgeführt und die Bundles installiert und ausgeführt werden.

Dazu müssen wir die Jar-Datei finden, mit der wir die Equinox-Konsole von unserer Kommandozeile aus starten können. Die Datei ist im Eclipse-Ordner unter Plug-ins zu finden und heißt derzeit „org.eclipse.osgi_3.6.0.v20100517.jar“ (je nach Version, die Datei fängt aber immer mit „org.eclipse.osgi_“ an). Damit kann die Equinox-Konsole nun gestartet werden. Dazu gehen wir in der Kommandozeile in den Ordner, in der sich diese Jar-Datei befindet und geben folgenden Befehl ein:

```
D:\Program Files\eclipse\plugins>java -jar org.eclipse.osgi_3.6.0.v20100517.jar
-console -clean
```

Das Argument **-console** führt dazu, dass das OSGi Framework direkt in der Kommandozeile geöffnet wird, **-clean** sorgt dafür, dass beim Start des Frameworks die Bundle Caches geleert werden. Dadurch ist sichergestellt, dass alle eventuell installierten Bundles aus dem System entfernt werden.

Durch das Aufrufen der Jar-Datei wird nun die Equinox-Konsole gestartet und kann wie aus Eclipse gewohnt verwendet werden.

Jetzt installieren wir alle drei Bundles:

```
osgi> install file:d:\temp\bundles\plugins\de.hs_osnabrueck.producer_1.0.0.20100
9290953.jar
Bundle id is 1

osgi> install file:d:\temp\bundles\plugins\de.hs_osnabrueck.consumer_1.0.0.20100
9290953.jar
Bundle id is 2

osgi> install file:d:\temp\bundles\plugins\de.hs_osnabrueck.producer.impl_1.0.0.
201009290953.jar
Bundle id is 3
```

Nun sind die Bundles installiert und können gestartet werden:

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.6.0.v20100517
1       INSTALLED   de.hs_osnabrueck.producer_1.0.0.201009290953
2       INSTALLED   de.hs_osnabrueck.consumer_1.0.0.201009290953
3       INSTALLED   de.hs_osnabrueck.producer.impl_1.0.0.201009290953

osgi> start 1

osgi> start 2
Consumer: Service anfordern...

osgi> start 3
Producer: Service registrieren...
Hello World!

osgi> stop 3
Good Bye World!
```

Welche Bundles wann gestartet werden, spielt dabei keine Rolle, da wir durch den implementierten **Service** Tracker dynamisch auf einen verfügbaren Service reagieren können.

6 Wichtige OSGi-Kommandos

Hier sind einige der wichtigsten Kommandos der Equinox-Konsole aufgeführt. Sobald ein Plug-in-Projekt ausgeführt wird, wird diese Konsole automatisch gestartet.

Kommando	Beschreibung
shutdown	das OSGi Framework wird heruntergefahren; der Zustand aller installierten Bundles wird gespeichert
exit	beendet die virtuelle Maschine unmittelbar
close	das OSGi Framework wird heruntergefahren und die virtuelle Maschine beendet
[i]nstall <URL>	installiert das Bundle mit der angegebenen URL; die URL kann z.B. file:/<Pfad zur Bundle> sein
[un]install (id || location || name)	deinstalliert angegebene(s) Bundle(s)
[sta]rt (id || location || name)	startet angegebene(s) Bundle(s)
[sto]p (id || location || name)	stoppt angegebene(s) Bundle(s)
[r]efresh (id || location || name)	löst Abhängigkeiten des/der angegebenen Bundle(s) neu auf
[up]date (id || location || name)	aktualisiert angegebenes Bundle; bei 'update *' werden alle installierten Bundles aktualisiert
ss	zeigt eine kurze Version der Statusanzeige an (short status)
[s]tatus	zeigt detaillierte Informationen zu installierten Bundles und Services an
[b]undle (id || location || name)	zeigt detaillierte Informationen über das/die angegebene(n) Bundle(s) an
bundles	zeigt detaillierte Informationen zu installierten Bundles an

7 Weiterführende Links und Literatur

- Die OSGI Service Platform – Eine Einführung mit Eclipse Equinox

  Buch von Gerd Wütherich, Nils Hartmann, Bernd Kolb und Matthias Lübken; sehr detaillierte Beschreibung des Themas; zu jedem Kapitel ein Tutorial

- http://www.osgi.org/ (englisch)

  Internetauftritt der OSGi Alliance; hier sind die aktuellen OSGi-Spezifikationen zu finden (Spezifikationen sollten erst gelesen werden, wenn sich bereits in das Thema eingearbeitet wurde)

- http://de.wikipedia.org/wiki/OSGi

  kurze Einführung in das Thema; relativ oberflächlich

- http://it-republik.de/jaxenter/artikel/Das-OSGi-Framework-2221.html

  „Das OSGi Framework“: sehr guter Artikel als Einführung

- http://it-republik.de/jaxenter/serien/?s=19

  „OSGi in kleinen Dosen“: mehrere sehr gute Artikel; tiefere Einführung in OSGi mit Tutorial

- http://www.vogella.de/articles/OSGi/article.htm (englisch)

  „OSGi with Eclipse Equinox“: sehr kurze Einführung mit Tutorials

- http://www.ralfebert.de/rcpbuch/osgi/

  „Plug-in basierte Entwicklung mit OSGi“: Einführung in das Thema mit Tutorial

- http://wiki.fernuni-hagen.de/eclipse/index.php/Hauptseite

  Eclipse-Wiki: viele hilfreiche Artikel, besonders:

  - http://wiki.fernuni-hagen.de/eclipse/index.php/Plug-ins

    Artikel über Plug-ins (Aufbau, Lebenszyklus etc.)