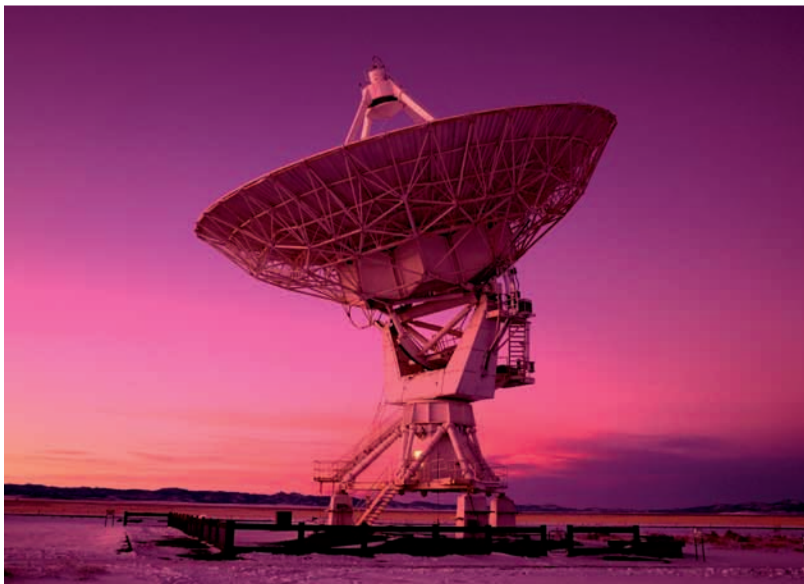
Unter der Haube, Teil 1: Ein Blick auf die Eclipse Runtime

Was Eclipse im Innersten zusammenhält

VON MARTIN LIPPERT

Eclipse ist nicht nur eine reichhaltige Plattform, auf deren Basis Plug-ins oder komplette Rich-Client-Anwendungen implementiert werden können. Unter der Haube dieser Plattform verbergen sich viele interessante Ideen, Entwürfe, Konstruktionen und Entwurfsmuster. Sie können über die Plattform hinaus Anregungen geben, wie Softwaresysteme gestaltet und implementiert werden können. In dieser Reihe werden wir einen Blick auf ausgewählte Elemente der Eclipse-Entwicklung werfen und zeigen, welche Prinzipien sich dahinter verbergen, wie diese Bestanteile der Plattform aufgebaut sind und wie sich solche Konstruktionen auch auf andere Situationen übertragen lassen. Der erste Teil dieser Reihe zeigt, wie die Runtime der Eclipse-Plattform aufgebaut ist, was OSGi damit zu tun hat und warum OSGi nicht alles ist.



Im Gegensatz zu „normalen" Java-basierten Anwendungen bestehen Eclipse-basierte Anwendungen grundsätzlich aus einer Menge von so genannten Plug-ins. Kein Element der Eclipse-Platform hat eine ähnlich grundsätzliche Bedeutung wie das Konzept der Plug-in-basierten Anwendungsentwicklung. Plug-ins erlauben es, Eclipse-basierte Anwendungen aus Komponenten zusammenzusetzen. Dazu besitzen Plug-ins einerseits explizit definierte Schnittstellen und Abhängigkeiten zu anderen Plug-ins und definieren andererseits Erweiterungspunkte, an denen ihre Funktionalität durch andere Plug-ins erweitert werden kann, sowie Erweiterungen, mit denen sie die Funktionalität anderer Plug-ins erweitern.

Zur Verfügung gestellt wird das Konzept der Plug-ins von der Eclipse Runtime, die damit das grundlegende Element der Eclipse-Plattform darstellt. Aber wie realisiert die Runtime von Eclipse, dass Anwendungen aus Plug-ins zusammengesetzt werden können, dass tatsächlich nur die definierten Abhängigkeiten existieren, alle Erweiterungen zu einem Erweiterungspunkt gefunden sowie noch zusätzlich Plug-ins dynamisch zur Laufzeit installiert und deinstalliert werden können?

Der Kern: OSGi

Im Kern der Eclipse Runtime arbeitet seit der Version 3.0 eine Implementierung

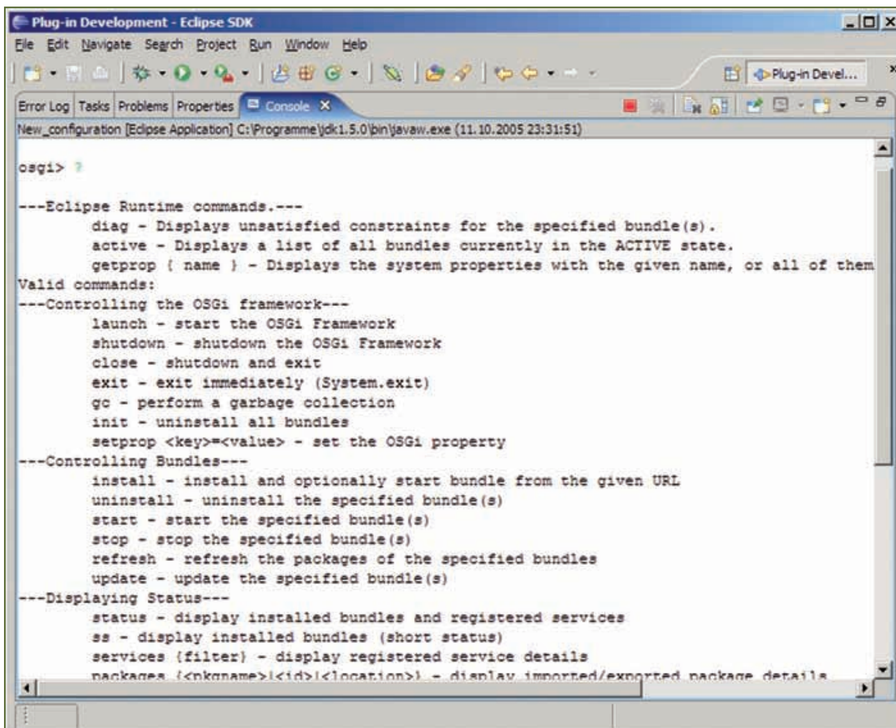


Abb. 1: Der Help Output (nach Eingabe von „?") der OSGi-Konsole informiert über die verfügbaren Kommandos.

der OSGi-Spezifikation (OSGi steht für Open Service Gateway Initiative – www.osgi.org). Sie stammt ursprünglich aus dem Bereich der Softwareentwicklung für eingebettete Systeme und diente dort dazu, einen Standard für Komponenten zu definieren. Innerhalb der Runtime von Eclipse arbeitet eine Implementierung der OSGi-Spezifikation, die ursprünglich von IBM stammt und dem Eclipse-Projekt gestiftet wurde.

Die Einscheidung, mit der Version 3.0 von Eclipse die Runtime auf eine OSGi-Implementierung aufzubauen, beruhte damals auf der Tatsache, dass viele der Features, die für die Eclipse Runtime der Version 3.0 gewünscht waren, von OSGi-Implementierungen bereits fertig zur Verfügung gestellt werden konnten. Dazu zählten vor allem die dynamischen Eigenschaften der OSGi-Implementierungen, die es erlauben, einzelne Komponenten dynamisch zur Laufzeit des Systems zu installieren und zu aktivieren als auch zu deaktivieren und zu deinstallieren. Solche Möglichkeiten der alten selbst implementierten Runtime hinzuzufügen hätte erheblichen Aufwand bedeutet und in einer proprietären Lösung geendet.

Ein OSGi-basiertes System besteht aus einer Menge von Komponenten, in OSGi auch Bundles genannt. Ein Bundle besteht dabei im Grunde genommen aus zwei Teilen: den deklarativen Meta-Daten, die das Bundle beschreiben, und dem eigentlichen Code bzw. den Ressourcen, die das Bundle mitbringt. Die Meta-Daten eines Bundles werden in einem Manifest beschrieben. Neben generellen Informationen des Bundles (Name, Hersteller, Version etc.) wird im Manifest auch deklariert, welche anderen Bundles benötigt werden und welches API anderen Bundles zur Verfügung gestellt wird.

Die OSGi-Implementierung ist dafür zuständig, dass die passenden Bundles gefunden werden, dass der Lebenszyklus der Bundles korrekt verwaltet wird (dies ist für das dynamische Installieren und Deinstallieren von besonderer Bedeutung) und dass nur die deklarierten Abhängigkeiten zwischen Bundles zur Laufzeit auch tatsächlich existieren. Die OSGi-Implementierung von Eclipse realisiert dies, indem der Code und die Ressourcen eines Bundles jeweils von einem eigenen Classloader geladen werden. Abhängigkeiten zwischen Bundles werden über einen Delegationsmechanismus zwischen den Classloadern der Bundles realisiert. Dieser recht ausgefeilte Delegationsmechanismus sorgt dafür, dass zur Laufzeit nur die deklarieren Abhängigkeiten existieren und Klassendefinitionen oder Ressourcen anderer Bundles nicht gefunden werden, solange eine entsprechende Abhängigkeit nicht explizit definiert wurde.

Tipp: Normalerweise bekommt man als Plug-in- oder RCP-Entwickler nur sehr wenig von der OSGi Runtime mit. Man kann allerdings eine Eclipse-Anwendung mit der Option *-console* starten. Daraufhin startet der OSGi-Kern der Eclipse Runtime eine Konsole, mit der zur Laufzeit beispielsweise der Status aller installierten Bundles überwacht werden kann, neue Bundles installiert oder alte beendet und deinstalliert werden können. Damit bekommt man tatsächlich einmal ein Stück der OSGi Runtime zu Gesicht und kann direkt mit ihr interagieren (Abb. 1).

Der Kern von Eclipse ist eine vollwertige OSGi-Implementierung. Deshalb kann diese auch unabhängig von anderen Eclipse-Komponenten isoliert als eigenständige OSGi-Umgebung verwendet werden. Wer also eine Alternative zu anderen bekannten OSGi-Implementierungen wie beispielsweise Knopflerfish (www.knopflerfish.org) oder Oscar (oscar.objectweb.org) sucht, sollte auch einen Blick auf die Eclipse-Implementierung werfen. Eine kurze Anleitung zur Standalone-Nutzung der Eclipse-OSGi-Implementierung findet sich unter www.eclipse.org/equinox/quickstart.html.

Bundles vs. Plug-ins

Während innerhalb der OSGi-Welt eigentlich nur Bundles bekannt sind, sprechen wir auf Ebene von Eclipse-Anwendungen meist von Plug-ins. Wie spielen nun Eclipse-Plug-ins und OSGi Bundles zusammen?

Im Kern der Eclipse Runtime läuft eine OSGi-Implementierung, die prinzipiell nur das Konzept von OSGi Bundles „versteht". (Das erlaubt es, in einer Eclipse-Anwendung auch gängige OSGi-Bundles zu verwenden, die nicht speziell für die Eclipse-Plattform implementiert wurden.) Deshalb sind Eclipse-Plug-ins

standardmäßig als OSGi Bundles definiert. OSGi-konforme Plug-ins erkennt man dadurch, dass sie eine Manifest-Datei besitzen, in der OSGi-konform die Meta-Daten des Plug-ins/Bundles deklariert werden. Erzeugt man mit Eclipse ein neues Plug-in-Projekt, wird deshalb neben der klassischen *plugin.xml*-Datei auch eine solche Manifest-Datei erzeugt. Lediglich wenn man im New-Plugin-Projekt-Wizard explizit den Haken von der Checkbox „Create an OSGi bundle manifest" entfernt, werden keine Manifest-Datei erzeugt und das Plug-in wie ein altes, mit der Eclipse-Version 2.1 implementiertes Plug-in behandelt (s.u.).

Bearbeitet werden die Inhalte beider Dateien allerdings mit dem altbekannten Plug-in-Editor, mit dem in der Vergangenheit auch schon die *plugin.xml*-Dateien recht komfortabel editiert werden konnten. Die Bereiche des Plug-in-Editors, die sich mit den Abhängigkeiten und APIs des Plug-ins beschäftigen, legen ihre Informationen automatisch in der Manifest-Datei ab.

Extension Points und Extensions

In den Manifest-Dateien von Plug-ins/Bundles werden allgemeine Meta-Daten deklariert. Das Plug-in-Konzept von Eclipse geht allerdings über das einfache Komponentenmodell von OSGi hinaus. Plug-ins können Erweiterungspunkte (Extension Points) und Erweiterungen (Extensions) für solche Erweiterungspunkte defineren. Plug-ins in Eclipse sind also OSGi Bundles, die um ein Erweiterungskonzept ergänzt wurden.

Diese Ergänzung des OSGi-Kerns um die „Erweiterbarkeit" von Plug-ins spiegelt sich auch im technischen Aufbau der Runtime selbst wieder. Im Kern der Eclipse Runtime verrichtet eine reine OSGi-Implementierung ihren Dienst. Die befindet sich im so genannten System-Bundle *org.eclipse.osgi*. Das Erweiterungs-Framework wird durch das Bundle *org.eclipse.core.runtime* dem OSGi-Kern hinzugefügt und läuft aus Sicht des OSGi-Kerns als normales OSGi Bundle – wie jedes andere Bundle auch. Lediglich eine Option legt fest, dass beim Start der Runtime nicht nur das OSGi-Framework gestartet werden muss, sondern auch das *org.eclipse.core.runtime* Bundle automatisch startet. Dieses Bundle sorgt wiederum dafür, dass die eigentliche Eclipse-Anwendung (beispielsweise die IDE Workbench), die als Extension definiert wird, gestartet wird.

Auch bei der Definition von Plug-ins ist diese Trennung erkennbar. Während die allgemeinen Meta-Daten eines Plug-ins sowie die Abhängigkeiten zwischen Plug-ins und die entsprechenden APIs eines Plug-ins in der Manifest-Datei Bundle-konform deklariert werden, befinden sich die Erweiterungsdeklarationen in der *plugin.xml*-Datei eines jeden Plug-ins.

Um es dem Plug-in-Entwickler möglichst einfach zu machen, wird auch die *plugin.xml*-Datei mit dem Plug-in-Editor bearbeitet. Die entsprechenden Teile des Plug-in-Editors, die sich mit Erweiterungspunkten und Erweiterungen befassen, greifen dazu auf die *plugin.xml*-Datei zurück und modifizieren diese entsprechend den Einstellungen des Entwicklers.

Alte Plug-ins – neue Runtime?

Die Eclipse Runtime basiert erst seit der Version 3.0 auf einem OSGi-Kern. In der Eclipse-Version 2.1 war die Runtime noch komplett selbst implementiert und es existierte keine Trennung in OSGi Bundles und Plug-ins. Folgerichtig besaßen Plug-ins für Eclipse 2.1 keine Manifest-Dateien. Allgemeine Meta-Daten sowie Abhängigkeiten und APIs von Plug-ins wurden damals komplett in der *plugin.xml*-Datei definiert (neben den Extension Points und Extensions, siehe Listing 1).

Und da Abwärts-Kompatibilität bei der Eclipse-Entwicklung groß geschrieben wird, ergab sich für die neue Runtime schnell eine sehr bedeutende Anforderung: Alte Plug-ins müssen unverändert auch auf Basis der neuen Runtime laufen können.

Alte Plug-ins müssen unverändert auch auf Basis der neuen Runtime laufen können.

Anzeige

Um diese Forderung zu erfüllen und trotzdem eine komplett neue Runtime einsetzen zu können, haben sich die Entwickler der Runtime für eine recht geschickte Strategie entschieden: Sie realisierten die neue Runtime auf Basis von OSGi, ohne sich allzu sehr von der Schnittstelle der alten Runtime beeinflussen zu lassen und implementierten einen speziellen Compatibility Layer, der die alte Schnittstelle der Runtime auf die neue Implementierung abbildet. Zu finden ist dieser Compatibility Layer im Plug-in *org.eclipse.core .runtime.compatibility*.

Aber wie können alte Plug-ins diesen Compatibility Layer nutzen? Sie definieren doch in der Regel einfach eine Abhängigkeit auf *org.eclipse.core.runtime*, dem kompletten Runtime-Plug-in in Eclipse 2.1. Und wie kann die neue Runtime überhaupt die Abhängigkeiten dieser alten Plug-ins identifizieren, wenn für alte Plug-ins keine Manifest-Datei existiert?

Auch hier bedient sich die neue Runtime eines einfachen Tricks. Findet sie ein Plug-in, welches noch für die Version 2.1 definiert wurde, erzeugt die aus der *plugin. xml*-Datei dynamisch eine passende Manifest-Datei. In diese Manifest-Datei hinein werden alle Daten aus der *plugin.xml* extrahiert, die das OSGi-Framework von einem guten Plug-in bzw. Bundle erwartet. Darüber hinaus generiert die Runtime in die Manifest-Datei auch eine zusätzliche Abhängigkeit zum Compatibility-Plug-in *org.eclipse.core.runtime.compatibility* hinein.

Diese Generierung kann man sehr gut beobachten, wenn man mit einer Eclipse 3.1-Installation ein 2.1-kompatibles Plug-in implementiert und eine Runtime Workbench startet. Im Workspace-Verzeichnis wird die passend generierte Manifest-Datei abgelegt unter: *.metadata\.plugins\org .eclipse.pde.core\<name der launch configuration>\org.eclipse.osgi\manifests* (Listing 2). Mit einem solchen Manifest könnten dann die entsprechenden Definitionen in der *plugin.xml*-Datei entfallen und nur noch die Extension-Point- und Extension-Definitionen übrig bleiben (Listing 3). Interessant ist diese Konstruktion vor allem auch deshalb, weil sie zeigt, dass mit einer guten Strategie selbst große Umstrukturierungen abwärts kompatibel durchgeführt werden können.

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="org.nonosgi.plugin"
  name="Plugin Plug-in"
  version="1.0.0"
  provider-name=""
  class="org.nonosgi.plugin.PluginPlugin">

  <runtime>
    <library name="plugin.jar">
      <export name="*"/>
    </library>
  </runtime>

  <requires>
    <import plugin="org.eclipse.ui"/>
  </requires>

  <extension
      point="org.eclipse.ui.views">
    <category
        name="Sample Category"
        id="org.nonosgi.plugin">
    </category>
    <view
        name="Sample View"
        icon="icons/sample.gif"
        category="org.nonosgi.plugin"
        class="org.nonosgi.plugin.views.SampleView"
        id="org.nonosgi.plugin.views.SampleView">
    </view>
  </extension>
</plugin>
```

Listing 2

```
Manifest-Version: 1.0
Generated-from: 1128973238239;type=2
Bundle-ManifestVersion: 2
Bundle-Name: Plugin Plug-in
Bundle-SymbolicName: org.nonosgi.plugin;
                                        singleton:=true
Bundle-Version: 1.0.0
Bundle-ClassPath: plugin.jar
Bundle-Activator: org.eclipse.core.internal.
                         compatibility.PluginActivator
Bundle-Localization: plugin
Export-Package: org.nonosgi.plugin,
 org.nonosgi.plugin.views
Require-Bundle: org.eclipse.ui,
 org.eclipse.ui.ide;resolution:=optional,
 org.eclipse.ui.views;resolution:=optional,
 org.eclipse.ui.editors;resolution:=optional,
 org.eclipse.jface.text;resolution:=optional,
 org.eclipse.ui.workbench.texteditor;resolution:
                                            =optional,
 org.eclipse.core.runtime.compatibility,
 org.eclipse.core.runtime;bundle-version="2.1"
Eclipse-AutoStart: true
Plugin-Class: org.nonosgi.plugin.PluginPlugin
```

Listing 3

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>

  <extension
     point="org.eclipse.ui.views">
   <category
      name="Sample Category"
      id="org.nonosgi.plugin">
   </category>
   <view
      name="Sample View"
      icon="icons/sample.gif"
      category="org.nonosgi.plugin"
      class="org.nonosgi.plugin.views.SampleView"
      id="org.nonosgi.plugin.views.SampleView">
   </view>
 </extension>

</plugin>
```

Ausblick

Wer denkt, die Runtime von Eclipse sei fertig und unterliege in der Zukunft keinen Veränderungen mehr, der irrt. Derzeitige Planungen deuten darauf hin, dass sowohl der OSGi-Kern weiterentwickelt und als Referenzimplementierung für das Release 4 der OSGi-Spezifikation dienen wird als auch weitere Teile der Runtime herausgelöst werden und somit isoliert verwendet werden können. Besonders interessant wird dies beispielsweise für das Erweiterungs-Framework, welches dann auch ohne OSGi Runtime beispielsweise in einer Serverumgebung oder jeder anderen Java-Anwendung einsetzbar wäre.

Abschließend danke ich Bernd Kolb für das Feedback zu diesem Artikel. Weiteres Feedback und Anregungen, auch für zukunftige Ausgaben dieser Reihe, sind jederzeit willkommen unter: martin.lippert@it-agile.de.

Martin Lippert ist Senior-IT-Berater bei der it-agile GmbH. Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und die Eclipse-Technologie.