Mit Eclipse 3 dynamische Plug-ins implementieren

Neustart?!? Warum?

VON MARTIN LIPPERT UND BERND KOLB

Wer mit der Version 2 von Eclipse gearbeitet hat, wird sich noch daran erinnern, dass man nach dem Installieren oder Deinstallieren eines Plug-ins die komplette IDE neu starten musste. Zwar passierte dies weitgehend automatisch (wenn man den Update-Mechanismus von Eclipse nutzte), aber bei großen Workspaces und sehr vielen Plug-ins in einer Installation konnte das schon einmal recht lange dauern. Mit der Einführung der OSGi-basierten Runtime in Eclipse 3.0 ist es möglich, Plug-ins dynamisch zur Laufzeit zu installieren und zu deinstallieren, ohne dass die laufende Anwendung neu gestartet werden muss. In diesem Artikel zeigen wir, welche Teile der Eclipse-IDE bereits mit dynamischen Plug-ins umgehen können und was der Plug-in-Entwickler beachten muss, um mit den dynamischen Fähigkeiten der Runtime korrekt umzugehen.

Mit der Version 3.0 basiert die Eclipse Runtime auf einem OSGi-Kernel (siehe auch OSGi-Artikel ab Seite 65). Dieser Kernel erlaubt es prinzipiell, dass Plug-ins dynamisch zur Laufzeit installiert und deinstalliert werden. Damit wird es möglich, neue Plug-ins zu installieren, ohne die laufende Anwendung (bspw. die Eclipse IDE) neu zu starten. Das dynamische Installieren von Plug-ins ließe sich beispielsweise auch dazu nutzen, um RCP-Anwendungen per URL zu starten, indem man in einer URL einfach die Update-Site der RCP-Anwendung angibt, Eclipse die RCP-Anwendungn installiert und dann anschließend startet, ohne dass zwischendurch die komplette Workbench neu gestartet werden muss.

Fast noch wichtiger als das dynamische Installieren von Plug-ins ist die Möglichkeit, Plug-ins auch dynamisch zur Laufzeit zu deinstallieren. Gerade für sehr große Installationen mit vielen hundert oder tausend Plug-ins kann es aus Speicherplatzgründen sehr nützlich sein, nicht mehr benötigte Plug-ins zu deaktivieren oder komplett zu deinstallieren – wobei man das Deinstallieren in großen Eclipse-basierten Anwendungen nur selten auf Ebene einzelner Plug-ins, sondern eher auf Ebene kompletter Features nutzen wird.

Die Runtime von Eclipse 3.0 stellt dazu die nötige Infrastruktur und Funktionalität zur Verfügung. Die neue Runtime implementiert daher einen umfangreichen Lebenszyklus für Plug-ins, der es erlaubt, Plug-ins zu installieren und zu deinstallieren, ohne die Runtime selbst neu zu starten. Damit ist die Grundvoraussetzung für dynamische Plug-ins geschaffen. Allerdings müssen sich Plug-ins in diesen Lebenszyklus einfügen und getreu diesem Lebenszyklus implementiert werden. Nur so kann die Runtime ihre grundsätzlichen Möglichkeiten ausspielen und Plug-ins tatsächlich mit den gewünschten Effekten (de)installieren. Schauen wir uns aber im Detail an, was das für ein Plug-in bedeutet.

Dynamische Plug-ins

Einerseits muss sich ein Plug-in selbst „gutartig“ gegenüber seinem dynamischen Lebenszyklus verhalten. „Gutartig“ bedeutet in diesem Fall, dass es dafür vorgesehen sein muss, sowohl dynamisch installiert als auch deinstalliert zu werden. Das dynamische Installieren von Plug-ins ist in der Regel ohne echten Zusatzaufwand für ein Plug-in realisierbar. Das liegt hauptsächlich daran, dass die grundsätzliche Architektur des Plug-in-Mechanismus in Zusammenarbeit mit dem Extension-Mechanismus von Eclipse schon seit der ersten Version darauf ausgelegt war, dass Plug-ins erst dann geladen werden, wenn sie wirklich gebraucht werden. Dazu werden die wesentlichen Informationen eines Plug-ins in der *plugin.xml*-Datei abgelegt. Da in der *plugin.xml*-Datei vor allem auch die Extensions definiert sind, die durch das Plug-in dem System hinzugefügt werden, können diese Informationen ausgewertet werden, ohne dass das Plug-in selbst aktiv wird. Die Workbench beispielsweise definiert einen Extension Point für Views und zeigt in einem Menü an, welche Views alle zur Verfügung stehen. Um diese Auswahl anzuzeigen, benötigt die Workbench le-

diglich die Extension-Definitionen aller Views, die durch Plug-ins hinzugefügt werden. Die Plug-ins selbst (ihr Code) werden dazu weder geladen noch aktiviert. Erst wenn eine View durch den Benutzer ausgewählt wird (bzw. durch die ausgewählte Perspektive angezeigt werden soll), aktiviert die Runtime das Plug-in, welches die View implementiert, und lädt den passenden Code. Mit diesem Mechanismus erreicht die Runtime, dass nur die Plug-ins aktiviert und geladen werden, die innerhalb der Anwendung auch tatsächlich durch den Anwender benötigt werden.

Bis einschließlich Version 2 wurden die *plugin.xml*-Informationen beim Start der Runtime gelesen und im Speicher abgelegt. Nach dem Start der Anwendung hinzugefügte Plug-ins wurden also nicht mehr berücksichtigt. Dies hat sich mit der neuen Runtime verändert, sie kann diese Informationen auch zur Laufzeit aktualisieren. Dies ist aber für das Plug-in, welches die Extension implementiert, nicht von Interesse. Stattdessen muss das Plug-in, welches den Extension Point verarbeitet, mit neuen oder entfernten Extensions umgehen, aber dazu mehr im nächsten Abschnitt.

Das dynamische Deinstallieren war im Gegensatz dazu bisher nicht durch die Runtime von Eclipse vorgesehen. Deshalb sind viele Plug-ins nicht per se so implementiert worden, dass sie auch zur Laufzeit deaktiviert und deinstalliert werden können. Wird ein Plug-in zur Laufzeit deaktiviert und deinstalliert, müssen beispielsweise alle während der Lebenszeit des Plug-ins angeforderten Ressourcen wieder freigegeben werden. Geschieht dies nicht, kann die Java-Laufzeitumgebung die entsprechenden Objekte und Klassen nicht per Garbage Collection aus der VM entfernen und das Plug-in selbst ist nicht mehr aktiv, aber noch vorhandene Objekte und Ressourcen verbleiben in der Java-VM und können weiterhin von anderen Objekten aufgerufen werden. Hat sich das Plug-in z. B. als *IResourceChangeListener* bei der Workbench registriert, muss es sich bei der Deinstallation auch korrekt wieder bei der Workbench abmelden.

Der entsprechende Code kann in den Hook-Methoden der Plug-in-Klasse implementiert werden. Dort befindet sich sowohl eine *start*- als auch eine *stop*-Methode. Die *stop*-Methode wird gerufen, wenn die Eclipse Runtime das Plug-in deaktiviert. Dies passiert auch immer dann, wenn das Plug-in deinstalliert wird.

```
/**
 * This method is called when the plug-in is stopped
 */
public void stop(BundleContext context) throws
                                                Exception {
  super.stop(context);
  plugin = null;
  resourceBundle = null;

  // hier Code einfügen, um Plug-in-spezifische
  // Ressourcen freizugeben
}
```

Erzeugt man ein neues Plug-in mit dem Plug-in Wizard von Eclipse, wird eine entsprechende Methode bereits in die Plug-in-Klasse, die in der Regel von *AbstractUIPlugin* erbt, generiert. Wichtig ist, dass man in einem solchen Fall zunächst die Implementierung aus der Oberklasse rufen sollte, da dort ebenfalls wichtige Aufräumarbeiten stattfinden. Zu den Ressourcen, die ein Plug-in üblicherweise wieder freigeben sollte, zählen vor allem:

- genutzte Grafik-Objekte wie Fonts, Images und Colors
- *IAdapterFactory*-Objekte, welche beim AdapterManager registriert wurden
- *IResourceChangeListener*-Objekte, die bei der Workbench registriert wurden
- Datenbank-Verbindungen oder ähnliche „externe“ Ressourcen

Denn auch hier gilt: Der Garbage Collector von Java schützt nicht automatisch vor selbst programmierten Speicherlecks!!!

Dynamische Extensions

Nachdem wir uns angesehen haben, was ein Plug-in selbst mitbringen muss, um sich in einem dynamischen Umfeld prinzipiell gutartig zu verhalten, gehen wir einen Schritt weiter und nehmen den Extension-Mechanismus von Eclipse genauer unter die Lupe. Als Beispiel betrachten wir die Workbench der Eclipse IDE. Sie bietet einen Extension Point an, um beispielsweise Views hinzuzufügen. Nehmen wir weiterhin an, dass es ein Plug-in gibt,

welches einen View zu der Workbench hinzufügt, und dass der Benutzer diese View in der aktuellen Perspektive geöffnet hat.

Betrachten wir uns das Objektgeflecht, welches sich daraus ergibt, so hat die Workbench ein Objekt der speziellen View über den Extension-Point-Mechanismus von Eclipse erzeugt und an der Oberfläche eingebettet. Seit diesem Zeitpunkt hält die Workbench eine Objektreferenz auf diese View. Aber was passiert mit dieser Objektreferenz, wenn das Plug-in, welches die View implementiert, deinstalliert wird? Irgendwie muss die Workbench dafür sorgen, dass diese Objektreferenz auf die View gekappt wird und die View aus der Workbench-Oberfläche verschwindet. (Zusätzlich müssen natürlich auch alle von View Plug-in implementierten Actions, Menüeinträge und Ähnliches entfernt werden. Der Einfachheit halber beschränken wir uns hier aber auf die eigentliche View.) Aber wie erfährt die Workbench davon, dass das View Plug-in deinstalliert wird? Eine ähnliche Frage stellt sich für den Fall, dass ein neues Plug-in zur Laufzeit installiert wird und dieses Plug-in einer neuen View implementiert. Auch dann muss die Workbench davon Kenntnis erlangen, um beispielsweise die Auswahl der Views im Menü entsprechend anzupassen.

Wenn Plug-ins selbst Extension Points definieren und mit passenden Extensions arbeiten (wie beispielsweise die Workbench mit Views), nutzen sie die Extension Registry, um alle verfügbaren Extensions zu einem Extension Point abzufragen. In der Regel sieht der entsprechende Code ungefähr folgendermaßen aus:

```
IExtensionRegistry registry =
                        Platform.getExtensionRegistry();
IExtensionPoint point =
        registry.getExtensionPoint("org.test", "pointid");
IExtension[] extensions = point.getExtensions();

for (int i = 0; i < extensions.length; i++) {
  // create executable extensions or similar
}
```

Jetzt kann es passieren, dass sich die Informationen aus der Extension Registry zur Laufzeit der Anwendung verändern. Wird ein Plug-in, welches eine Extension zu diesem Extension Point implementiert, dynamisch installiert, taucht diese Extension beim nächsten Aufruf der *ExtensionRegistry* automatisch mit auf. Wird das Extension Plug-in deinstalliert, dann verschwindet ebenfalls die entsprechende Extension-Definition aus der Extension Registry.

Da das Plug-in, welches den Extension Point definiert, aber nicht wissen kann, wann ein Plug-in dynamisch installiert oder deinstalliert wird, muss es über entsprechende Veränderungen an der Extension Registry benachrichtigt werden. Die Extension Registry bietet genau für diesen Fall an, entsprechende Listener zu registrieren:

Beste Basis für dynamische Plug-ins

```
registry.addRegistryChangeListener
                        (new IRegistryChangeListener() {
  public void registryChanged
                        (IRegistryChangeEvent event) {
    // react on registry changes
  }
});
```

Die Methode *registryChanged* wird von der Extension Registry gerufen, sobald neue Extensions hinzukommen oder vorhandene wegfallen (durch das Installieren oder Deinstallieren von Plug-ins). Aufgabe des Plug-ins, welches den Extension Point anbietet, ist es nun, auf entsprechende Events zu reagieren. Das mitgelieferte Event-Objekt gibt Auskunft darüber, was sich an der Extension Registry verändert hat. Zunächst können wir die Delta-Objekte für den passenden Extension Point erfragen. Anschließend prüfen wir jedes Delta-Objekt darauf, ob es sich um eine neue oder eine weggefallende Extension handelt:

```
if (extensionDeltas[i].getKind() ==
                        IExtensionDelta.ADDED) {
  IExtension extension = extensionDeltas[i].
                        getExtension();
  // do something with new extension
}
else if (extensionDeltas[i].getKind() ==
                        IExtensionDelta.REMOVED) {
  IExtension extension = extensionDeltas[i].getExtension();
  // cut all references to objects from old extension
}
```

Bei entfernten Extensions ist es besonders wichtig, alle Referenzen auf ggf. schon erzeugte Objekte der Extension zu kappen. Nur dann kann die Java-VM alle entsprechenden Objekte aus dem Speicher entfernen und das Plug-in ist wirklich komplett aus der laufenden Anwendung deinstalliert.

Was ist bereits dynamisch?

Wir haben gesehen, dass die OSGi-basierte Runtime von Eclipse zwar die besten Voraussetzungen für dynamische Plug-ins liefert, aber die einzelnen Plug-ins ebenfalls ihren Beitrag dazu liefern müssen. Einer der wichtigsten Teile der Eclipse-IDE ist sicherlich die Workbench, die zentrale Extension Points für Views, Editors, Actions etc. anbietet. Da die Workbench in ihrer ursprünglichen Version nicht mit dem Blick auf dynamische Plug-ins implementiert war, musste sie relativ aufwendig an dynamische Plugins angepasst werden. In der Version 3.0 von Eclipse konnten diese Arbeiten nicht komplett abgeschlossen werden. Deshalb ist mit der Version 3.0 zwar das dynamische Installieren von Plug-ins größtenteils nutzbar, das Deinstallieren von UI Contributions zur Workbench ist in dieser Version allerdings noch nicht vollständig implementiert. Mit der Version 3.1 soll sich dies aber ändern.

Martin Lippert ist Senior-IT-Berater bei it-agile. Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und Eclipse-Technologie. Kontakt: martin.lippert@it-agile.de.

Bernd Kolb ist freiberuflicher Berater und Coach. Er ist (Mit-)Autor von diversen Artikeln sowie eines Buches und Sprecher auf Konferenzen. Seine Schwerpunkte liegen auf modellgetriebener Softwareentwicklung sowie Eclipse. Kontakt: b.kolb@kolbware.de.

Links & Literatur

[1] John Arthorne, Chris Laffra: Official Eclipse 3.0 FAQ, Addison-Wesley, 2004. Neben den vielen nützlichen Tipps und Tricks im Umgang mit Eclipse enthält dieses Buch auch einige, kleinere Abschnitte zu dynamischen Plug-ins.