Equinox erweitern und modifizieren

Getting Hooked on Equinox

>> MARTIN LIPPERT UND HEIKO SEEBERGER

Speaker @ jax

Quellcode auf CD

Viele verwenden Equinox als fertige OSGi Runtime, auf deren Basis vorhandene Anwendungen erweitert oder neu implementiert werden. Dazu kann Equinox in der Regel out-of-the-box verwendet werden. Was passiert aber, wenn diese Box nicht das leistet, was benötigt wird? Equinox bietet einen eleganten und wertvollen Mechanismus, um in solchen Situationen die Runtime selbst zu erweitern und zu modifizieren.

Equinox ist nicht nur Lieferant einer OSGi-Implementierung und erlaubt es damit, Anwendungen auf Basis von OSGi zu entwickeln. Die Equinox Runtime selbst ist mit dem Blick auf Erweiterbarkeit realisiert und bietet einen sehr mächtigen Mechanismus, um die OSGi Runtime zu erweitern oder zu modifizieren [1]. Zunächst soll ein Blick auf den Aufbau geworfen werden.

Aufbau der Equinox-OSGi-Implementierung

Bevor der Erweiterungsmechanismus von Equinox im Detail vorgestellt wird, soll der konzeptionelle Aufbau von Equinox betrachtet werden.

Equinox realisiert einerseits die drei OSGi Framework Layer: Der *Module Layer* ist dafür zuständig, das OSGi-Modul-Konzept in Form von Bundles umzusetzen sowie die Abhängigkeiten zwischen Bundles zu verwalten und aufzulösen. Dazu gehört auch das umfangreiche Delegationsmodell, mit dem die im Bundle-Manifest definierten Sichtbarkeiten von Packages zwischen Bundles realisiert werden. Der *Life Cycle Layer* kontrolliert den Lebenszyklus von Bundles und gestattet es, Bundles zur Laufzeit zu installieren, zu deinstallieren oder zu aktualisieren. Zusätzlich dazu realisiert dieser Layer auch einen Event-Mechanismus, über den OSGi das System über Zustandsänderungen benachrichtigt. Der dritte Layer ist der *Service Layer*. Dieser implementiert das OSGi-Servicekonzept, mit dem Bundles innerhalb einer Java Virtual Machine serviceorientiert zusammenarbeiten können.

Die Anbindung dieser Layer an eine konkret zugrunde liegende Plattform wird durch das Framework Adaptor API hergestellt. Der *FrameworkAdaptor* definiert das Format der Bundles, realisiert den Zugriff auf Ressourcen der Bundles über den Classloading-Mechanismus und verwaltet die Speicherung der persistenten Daten der OSGi Runtime.

Natürlich bringt Equinox eine Standardimplementierung des *FrameworkAdaptor*s mit. Dieser *BaseAdaptor* erlaubt es, durch so genannte Framework Extensions alle drei Bereiche (Bundle-Formate, Classloading und Persistent Storage) zu erweitern. Dazu bietet dieser *BaseAdaptor* einen umfangreichen

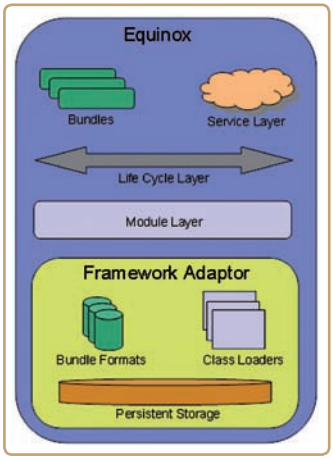

Abb. 1: Konzeptioneller Aufbau der Equinox-OSGi-Implementierung [2]

Hook-Mechanismus, der es sehr einfach erlaubt, an vordefinierten Stellen die Standardfunktionalität zu erweitern.

Hookable Adaptor

Vor Eclipse 3.2 war es nötig, für jegliche Anpassungen oder Erweiterungen einen eigenen *FrameworkAdaptor* zu implementieren, entweder komplett oder durch Erweiterung der gegebenen Standardimplementierung. Existierten mehrere, voneinander unabhängige Erweiterungen (also mehrere *FrameworkAdaptors*), konnte allerdings nur einer davon von Equinox verwendet werden (System Property *osgi.adaptor*), weil es konzeptionell nur einen *FrameworkAdaptor* zur Laufzeit geben kann.

Seit Eclipse 3.2 sieht der Standard-*FrameworkAdaptor*, der oben erwähnte *BaseAdaptor*, explizit Erweiterungsmöglichkeiten in Form von so genannten *Hooks* vor, weshalb er auch als *Hookable Adaptor* bezeichnet wird. Im Gegensatz zu selbst definierten *FrameworkAdaptors* kann es zur Laufzeit nun beliebig viele dieser Hooks geben. Sie werden alle vom *BaseAdaptor* entsprechend berücksichtigt. Es können also mehrere Erweiterungen (in Form von Hooks) gleichzeitig zum Einsatz kommen. Vom Equinox-Team wird ausdrücklich empfohlen, diesen Hook-Mechanismus zu verwenden und keinen eigenen *FrameworkAdaptor* mehr zu implementieren. Falls dies aufgrund von besonderen Anforderungen nicht möglich ist, sollte ein entsprechender Bug gemeldet werden.

Hook Interfaces

Welche verschiedenen Möglichkeiten bietet Equinox nun, um die Standardfunktionalität zu erweitern?

- Der *AdaptorHook* klinkt sich direkt in verschiedene Methoden des *FrameworkAdaptors* bzw. des *BaseAdaptors* ein. So kann z.B. beim Starten und Stoppen der OSGi Runtime eingegriffen oder die Behandlung von Laufzeitfehlern erweitert werden.
- Mit dem *BundleFileFactoryHook* ist es möglich, spezielle Formate für Bundles zu verwenden, die über den Equinox-Standard (JARs und Verzeichnisse) hinausgehen.
- Der *BundleFileWrapperFactoryHook* ermöglicht es, den Zugriff auf die Ressourcen eines Bundles zu wrappen, z.B. um Security zu implementieren, oder den Inhalt von einzelnen Ressourcen beim Laden gezielt zu manipulieren.
- Mit dem *BundleWatcher* kann der Lifecycle von Bundles verfolgt werden, wobei für jeden Statusübergang Start- und End-Events propagiert werden. Die Funktionalität ist jedoch ähnlich wie bei den OSGi *BundleListenern*.
- Der *ClassLoadingHook* dient dazu, das Classloading-Verhalten zu modifizieren. So kann zum Beispiel der Bytecode modifiziert werden, bevor eine Klasse in der Virtual Machine definiert wird.
- Der *StorageHook* wird für das Speichern und Laden von installierten Bundles verwendet.

Da es sich bei den Hooks um Erweiterungen der Standardfunktionalität der OSGi Runtime handelt, die zum Großteil im System Bundle *org.eclipse.osgi* implementiert ist, kommen so genannte *Framework Extensions* zum Einsatz. Eine Framework Extension ist ein Fragment Bundle zum System Bundle, d.h. der Fragment Host ist *system.bundle* bzw. *org.eclipse.osgi*. Damit Fragment Extensions von Equinox verwendet werden, müssen sie über die System Property *osgi.framework.extensions* bekannt gegeben werden, als JAR vorliegen und im Dateisystem im gleichen Verzeichnis wie das System Bundle liegen.

Um Equinox einzelne Hooks bekannt zu machen, müssen sie über so genannte *HookConfigurators* bzw. deren *addHooks()*-Methode bei der *HookRegistry* registriert werden. Dabei ist es ein geläufiges Pattern, dass konkrete Hooks sowohl ein spezielles Hook Interface als auch *HookConfigurator* implementieren und sich somit selbst registrieren.

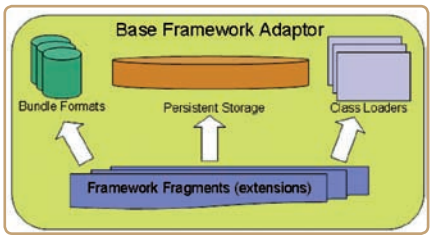

Abb. 2: Der Aufbau des Base Framework Adaptors [2]

Das führt zur nächsten Frage: Wie werden *HookConfigurators* von Equinox erkannt? Der eleganteste Weg führt über die Datei *hookconfigurators.properties*, die von Equinox im Root von Framework Extensions erwartet wird, und dort die Property *hook.configurators*. So kann eine Fragment Extension selbst angeben, welche *HookConfigurators* sie liefert. Über die System Properties *osgi.hook.configurators.include* bzw. *osgi.hook.configurators.exclude* können weitere bekannt gegeben bzw. vorhandene ausgeschlossen werden.

Ein Beispiel

Die Framework Extension *de.metafinanz.demo.equinox.extension* enthält mit dem *DemoHook* einen *AdaptorHook*, der beim Starten und Stoppen der OSGi Runtime Trace Messages auf der Konsole ausgibt. Weiterhin implementiert *DemoHook* auch *HookConfigurator*, wobei er sich selbst registriert und dabei ebenfalls eine Trace Message ausgibt. In Listing 1 sehen wir, wie mit der ersten Methode das *HookConfigurator* Interface implementiert wird, indem das eigene Objekt bei der übergebenen *HookRegistry* angemeldet wird. Die restlichen Methoden erfüllen das Interface *AdaptorHook*.

Um dieses Beispiel im Eclipse SDK auszuführen, ist es erforderlich, das System Bundle *org.eclipse.osgi* als Source in den Workspace zu importieren (im mitgelieferten Beispiel-Workspace bereits enthalten), damit die Framework Extension zur Laufzeit im selben Verzeichnis liegt wie das System Bundle (siehe oben). Zum Ausführen wird eine OSGi Framework Run Configuration (im mitgelieferten Beispiel-Workspace bereits enthalten) verwendet, welche die Workspace Bundles *org.eclipse.osgi* und *de.metafinanz.demo.equinox.extension* enthält und die System Property *osgi.framework.extensions* mit dem Wert *de.metafinanz.demo.equinox.extension*

belegt. Die passende *hookconfigurators.properties*-Datei ist ebenfalls schon fertig in der Framework Extension enthalten:

```
hook.configurators=de.metafinanz.demo.equinox.
                                extension.DemoHook
```

Existierende Erweiterungen

Wofür man diesen Mechanismus beispielsweise einsetzen kann, zeigen einige existierende Hook-Implementierungen:

- Built-In: In Equinox befinden sich bereits Hook-Implementierungen, die unter der Haube verwendet werden. Der Eclipse LazyStart Header im Bundle-Manifest wird beispielsweise durch den EclipseLazyStarter realisiert, der einen ClassLoadingStatsHook sowie einen AdaptorHook registriert und ein Bundle beim Zugriff auf dessen Klassen aktiviert. Weitere intern genutzte Hooks sind unter [3] beschrieben.
- Equinox Transforms [4]: Dieses Incubator-Projekt aus dem Equinox-Umfeld erlaubt es, mittels unterschiedlicher Techniken Ressourcen eines Bundles zu modifizieren, bevor auf sie zugegriffen wird. Die Modifizierungen können mit unterschiedlichen Techniken vorgenommen werden. Ein besonders attraktives Beispiel ist, die plugin.xml-Dateien mittels XSLT zu transformieren, um beispielsweise definierte Extensions je nach eingeloggtem Benutzer auszublenden. Damit könnte man elegant den Extension-Point-Mechanismus mit einem Autorisierungskonzept verbinden.
- J9 Class Sharing [5]: Die IBM J9 VM [6] für Java bietet einen Mechanismus an, bereits geladene Klassen in einem Cache abzulegen, um damit einerseits zwischen parallel laufenden VM-Instanzen die geladenen Klassen gemeinsam zu benutzen, um Speicher zu sparen. Andererseits müssen die Klassen nicht aufwändig geladen werden, wenn sie bereits im Cache liegen, was Performance-Vorteile bringt. Das normale J9 Class Sharing klinkt sich in den URLClassLoader ein und wirkt sich damit nicht auf eine Equinox-basierte OSGi-Anwendung aus, da dort spezielle ClassLoader zum Einsatz kommen. Mit passenden Hooks lässt sich aber das Classloading von Equinox so anpassen, dass dort die API der J9-VM angesprochen wird und so das Class Sharing auch für OSGi-Anwendungen nutzbar wird.
- Load-Time Aspect Weaving: Das Equinox-Aspects-Incubator-Projekt nutzt den Hook-Mechanismus von Equinox, um Aspect Weaving beim Laden der Klassen durchzuführen (Load-Time Weaving). Dort wird u.a. der ClassLoadingHook verwendet, um den Bytecode der geladenen Klassen per Aspect Weaving zu modifizieren. Zusätzlich wird der BundleFileWrapperFactoryHook genutzt, um durch das Weaving entstehende zusätzliche Abhängigkeiten zwischen Bundles mittels Modifikationen am Bundle-Manifest zu ermöglichen.
- Andere Class-File-Formate: Es gibt Situationen, in denen Class-Dateien nicht im üblichen Class-File-Format ausgeliefert werden. Stattdessen kommen spezialisierte Formate zum Einsatz.

Fazit

Zunächst sei noch einmal darauf hingewiesen, dass dieser Hook-Mechanismus nicht zum Standard für jede Anwendung werden sollte. In der Regel lassen sich die meisten Herausforderungen, auch wenn sie vielleicht auf den ersten Blick nach einem Classloading-Problem aussehen, durch die normalen OSGi- oder Equinox-Mechanismen lösen. Der Hook-Mechanismus sollte wirklich ausschließlich für Spezial-Lösungen herangezogen werden, die sich mit den OSGi- und Equinox-Bordmitteln nicht lösen lassen und gleichzeitig eine echte und vor allem OSGi-konforme Erweiterung der Runtime darstellen. Dann aber bietet der Hook-Mechanismus von Equinox einen beeindruckenden Reichtum an Möglichkeiten, der nicht zuletzt auch Spaß beim Ausschöpfen macht.

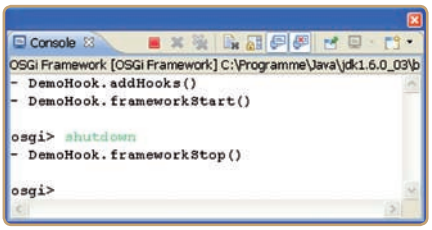

Abb. 3: Ausgabe des Beispiels

Listing 1

Die Hook-Implementierung

```
public class DemoHook implements HookConfigurator,
                                 AdaptorHook {

  public void addHooks(final HookRegistry hookRegistry) {
    hookRegistry.addAdaptorHook(this);
    System.out.println("- DemoHook.addHooks()");
  }

  public void frameworkStart(final BundleContext
                                 context) {
    System.out.println
                    ("- DemoHook.frameworkStart()");
  }
  public void frameworkStop(final BundleContext
                                 context) {
    System.out.println("- DemoHook.frameworkStop()");
  }

  // Other AdaptorHook methods are implemented empty
  // and not shown here!
}
```

Heiko Seeberger leitet die Market Unit Enterprise Architecture der metafinanz GmbH (www.metafinanz.de). Er erstellt seit etwa zehn Jahren Enterprise-Anwendungen mit Java, wobei sein aktueller Fokus auf Eclipse und AspectJ liegt. Kontakt: heiko.seeberger@metafinanz.de.

Martin Lippert ist Senior-IT-Berater bei der akquinet it-agile GmbH. Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und Eclipse-Technologie und ist Committer im Eclipse-Equinox-Incubator-Projekt. Kontakt: martin.lippert@akquinet.de.

>> Links & Literatur

[1] Informationen zum Aufbau der Equinox-Runtime: www.eclipsecon.org/2007/index.php?page=sub/&id=3762

[2] Chris Laffra, Thomas Watson, Matthew Webster: Getting Hooked on the Equinox Framework, IBM, EclipseCon 2007, Creative Commons Att. Nc Nd 2.5 license

[3] Beschreibung der Technik: wiki.eclipse.org/Adaptor_Hooks

[4] Equinox Transforms: wiki.eclipse.org/Equinox_Transforms

[5] J9 Class Sharing: www.ibm.com/developerworks/java/library/j-ibmjava4

[6] IBM J9 VM: wiki.eclipse.org/J9

[7] Equinox-Aspects-Incubator: www.eclipse.org/equinox/incubator/aspects/index.php