Finden und Gefunden werden

Classloading in Eclipse

Martin Lippert, lippert@acm.org, www.martinlippert.com

Mit der Version 3.0 liefert die Eclipse-Platform nicht nur eine umfangreiche Java-IDE aus, sondern auch eine umfassende Plattform zur Entwicklung von Rich-Client-Anwendungen: Die Eclipse Rich Client Platform (Eclipse-RCP). Neben den UI-fokussierten Rahmenwerken beinhaltet die Eclipse-RCP vor allem das Konzept der plugin-basierten Anwendungsentwicklung. Wesentliche Teile der Plugin-Architektur von Eclipse werden dabei von der Eclipse-Runtime realisiert. Die Runtime nutzt dazu ein sehr spezielles Classloading, um einerseits Plugins als separate, voneinander gekapselte Einheiten zu verwalten und andererseits zur Laufzeit nur die auch tatsächlich definierten Abhängigkeiten zwischen Plugins zu erzwingen. Allerdings hat dieses spezialisierte Classloading seine Tücken: Wer sich bisher nur äußerst selten mit einer *ClassNotFoundException* konfrontiert sah, wird sich durch die Eclipse-RCP schnell mit ihr anfreunden müssen. Nur zu häufig erscheint sie auf dem Bildschirm, gerade wenn Bibliotheken wie Hibernate oder JDO zum Einsatz kommen. Meist ist die Verwirrung darüber dann groß. Ich möchte in diesem Artikel zeigen, wie das Classloading in Eclipse funktioniert und wie typische Classloading-Probleme bei der Entwicklung mit der Eclipse-Rich-Client-Platform identifiziert und gelöst werden können.

Classloading in Java

Entgegen der weitläufigen Meinung werden Klassen in der Laufzeitumgebung einer Java-Anwendung nicht ausschließlich über ihren voll qualifizierten Namen (Name der Klasse plus Package-Bezeichner) eindeutig identifiziert. In Java-Laufzeitumgebungen werden Klassen von so genannten Classloadern geladen. Ein Classloader ist dafür verantwortlich, den Bytecode von benötigten Klassen im Classpath der Anwendung ausfindig zu machen und der Java-VM zur Verfügung zu stellen. Die Klasse wird dann innerhalb der VM eindeutig durch die Kombination aus ihrem voll qualifizierten Namen und dem Classloader, durch den sie geladen wurde, identifiziert.

Das bedeutet insbesondere, dass eine Klasse durchaus mehrfach in eine VM geladen werden kann. Sie muss dazu lediglich von unterschiedlichen Classloadern geladen werden. Obwohl die Classloader denselben Bytecode in die VM laden, unterscheidet die Java-VM die geladenen Klassen strikt voneinander. Weder ihre Class-Objekte sind identisch noch Objekte der verschiedenen Instanzen typkompatibel.

Gängige Java-Client-Applikationen machen von diesen Classloading-Features in der Regel wenig Gebrauch, da meist alle Klassen der Applikation von demselben Classloader geladen werden (dem Standard-Application-Classloader des JDK). Application-Server hingegen nutzen gerade diese Classloading-Mechanismen von Java intensiv, um beispielsweise Hot-Deployment zu realisieren oder verschiedene Anwendungen voneinander zu isolieren.

Dass das Laden von Klassen durch unterschiedliche Classloader auch für Verwirrung sorgen kann, zeigt sich anhand von *static*-Variablen. Der typische Java Entwickler geht davon aus, dass der Zugriff auf eine *static*-Variable einer Klasse in einer VM immer auf dieselbe Variable zugreifen wird; im Application-Server-Umfeld kann er durchaus böse Überraschungen erleben. Wird die Klasse durch Hot-Deployment (oder ähnliches) mehrfach von dem Application-Server angezogen, existieren auch mehrere Instanzen der static-Variable.

Classloading in Eclipse

Die Eclipse-Platform basiert auf der Idee der plugin-basierten Anwendungsentwicklung. Große Systeme werden dabei aus einer Menge kleiner Plugins an wohldefinierten Stellen („Extension Points“) zusammengesteckt. Auch das komplette Eclipse-SDK ist als eine solche Menge von Plugins realisiert.

Ein wichtiges Ziel plugin-basierter Entwicklung ist dabei, die einzelnen Plugins separat voneinander entwickeln und veröffentlichen zu können. Um das zu ermöglichen, bringen Eclipse-Plugins einerseits ihre eigenen Klassen mit und definieren andererseits explizit ihre Abhängigkeiten zu anderen Plugins. Beide Eigenschaften kombiniert führen dazu, dass im Prinzip jedes Plugin seinen eigenen Classpath definiert. So lassen sich typische Classpath-Konflikte (Library A arbeitet nur mit dem XML-Parser in Version 1.4, während Library B unbedingt Version 2.1 des gleichen Parsers erfordert) vermeiden.

Die Eclipse-Platform trennt die einzelnen Plugins voneinander, indem die Klassen eines jeden Plugins von einem eigens für dieses Plugin erzeugten Classloader geladen werden. Dadurch wird vermieden, dass sich Plugins gegenseitig stören und Seiteneffekte aufeinander ausüben. Natürlich kann ein Plugin trotzdem Klassen eines anderen Plugins verwenden. Dies ist aber nur dann möglich, wenn eine Abhängigkeit zwischen diesen Plugins definiert ist, und die Klassen vom „besitzenden“ Plugin als von außen zugreifbar („export“) werden. Ist dies nicht der Fall, erlaubt die Eclipse-Runtime nicht, dass ein Plugin die Klassen eines anderen Plugins benutzt.

In Eclipse 3.0 ist die Trennung von Plugins mittels unterschiedlicher Classloader durch den OSGi-Kernel der Runtime implementiert (siehe auch http://www.osgi.org).

Classloader-Hierarchien und Delegation

Zwischen unterschiedlichen Classloadern innerhalb einer VM können unterschiedliche Beziehungen herrschen. Klassischerweise werden Classloader in einer Hierarchie zueinander angeordnet. Soll eine Klasse von einem Classloader einer solchen Hierarchie geladen werden, wird immer zunächst der übergeordnete Classloader aufgefordert, die Klasse zu laden. Nur wenn dieser die Klasse nicht laden kann, kommt der nächst untergeordnete Classloader zum Zug. Wird die Hierarchie nicht explizit aufgebaut, wird immer der Classloader der Application als *parent* eines neuen Classloaders eingesetzt. Durch diese Hierarchie ist es beispielsweise nicht möglich, einen neuen Classloader zu erzeugen und die Klassen des JDK durch diesen ein weiteres Mal laden zu lassen (da die Klassen weiterhin von dem übergeordneten Classloader, dem Boot-Classloader der VM, gefunden werden).

Die Eclipse-Platform nutzt die hierarchische Beziehung zwischen Classloadern kaum. Zwar besitzt jeder Classloader eines Plugins einen übergeordneten Classloader, jedoch wird dieser explizit auf den Boot-Classloader der VM gesetzt[1]. Die Beziehungen zwischen Plugins werden durch einen Delegations-Mechanismus realisiert. Dieser erlaubt es einem Plugin, die Klassen eines verwendeten Plugins über dessen Classloader zu nutzen – nicht jedoch umgekehrt.

Versucht ein Plugin in Eclipse eine Klasse zu laden, passiert demnach folgendes:

1. Erst wird der Parent-Classloader gefragt, ob er die Klasse laden kann. Dies ist normalerweise der Boot-Classloader der Anwendung. Über diesen Classloader werden alle Klassen des bootClassPath angezogen (wenn nicht anders definiert sind dies die Klassen der JDK-Libraries).

2. Anschließend wird über den OSGi-Mechanismus festgestellt, ob die gesuchte Klasse aus einem von diesem Plugin referenzierten Plugin geladen werden kann oder ob das Plugin selbst diese Klasse zur Verfügung stellt. Im Erfolgsfall wird die Klasse dann geladen.

ClassNotFoundException

Während für eine einfache plugin-basierte Anwendung das spezielle Classloading den durchaus gewünschten Effekt der Isolation unterschiedlicher Plugins realisiert, kommt es häufig zu Schwierigkeiten, wenn zusätzliche Bibliotheken verwendet werden, die

[1] Mit der Option *osgi.parentClassloader* kann im config.ini-File der übergeordnete Classloader für alle Plugins umgesetzt werden, sowohl auf den Extension-Classloader der Java-VM, als auch auf den Application-Classloader der Eclipse-Anwendung. Siehe dazu auch http://dev.eclipse.org/newslists/news.eclipse.technology.equinox/msg00887.html

ihrerseits spezielle Annahmen über das Laden von Klassen machen. Bekannte Beispiele dafür sind:

- **Web- oder Application-Server:** Sie implementieren und/oder nutzen meinst selbst spezielle Classloader. Kombiniert man einen Web- oder Application-Server mit der Plugin-Runtime von Eclipse, kommen sich die verschiedenen Classloading-Ansätze gegenseitig in die Quere.
- **Client-J2EE-Container:** Sie nutzen häufig Reflection-Mechanismen, um unterschiedliche Implementationen dynamisch zu erzeugen, beispielsweise beim Lookup.
- **Persistenz-Frameworks:** Sie erzeugen meist Klassen per Reflection, wenn Objekte aus dem Persistenzmedium geladen werden. Dies ist auch bei der Standard-Serialisierung von Java der Fall.

Generell können alle Libraries (oder auch selbst implementierte Code-Teile) Probleme verursachen, die Klassen über Reflection (beispielsweise *Class.forName()*) nutzen.

Betrachten wir dazu ein Beispiel: Eine Persistenz-Library, wie im einfachsten Fall die Java-Serialisierung, wird von einem Plugin benutzt, um ein Geflecht von Objekten zu speichern. Das Objekt-Geflecht besteht nicht nur aus Objekten, deren Typen innerhalb des gleichen Plugins definiert wurden. Stattdessen beinhaltet das Objekt-Geflecht durchaus Typen, die nicht direkt von dem Plugin geladen werden können, aus dem heraus wir das Objekt-Geflecht speichern wollen. In der Regel klappt das *Speichern* solche Objektgeflechte noch relativ problemlos. Versuchen wir jedoch, die Objekte wieder zu laden, muss der Persistenzmechanismus entsprechende Objekte erzeugen. Dazu benötigt er die passenden Klassendefinitionen, die aber von dem Plugin aus, indem wir das Laden gerade angestoßen haben, nicht geladen werden können. Theoretisch können die Objekte, die wir laden wollen, schließlich von jedem beliebigen Classloader des Systems geladen worden sein. Wir bekommen eine *ClassNotFoundException*.

Klassen registrieren

Einige Implementierungen entsprechender Libraries erlauben es dem Benutzer, verwendete Klassen explizit bei der Library zu registrieren. Damit ist die Library nicht länger gezwungen, die Klassen dynamisch per Reflection zu erzeugen. Stattdessen kann sie die registrierten Class-Objekte direkt verwenden, um passende Objekte zu erzeugen. Dies ist zwar eine saubere Lösung für das Classloading-Problem, setzt aber voraus, dass die Library eine entsprechende API mitbringt und dass alle Klassen dort registriert werden, die durch die Library verwendet werden müssen (wenn auch nur indirekt). Im System kann dies zu dem relativ unschönen Umstand führen, dass man eine riesige Liste von Klassen dort registriert und trotzdem immer wieder in die Falle läuft, einzelne Klassen dort zu vergessen.

Resolve-Hooks

Einige Bibliotheken bieten Hook-Methoden an, die zum Auffinden von Klassendefinitionen genutzt werden können. Diese Hooks sind meist so vorimplementiert, dass sie über den normalen Classloading-Mechanismus die passende Klassendefinition suchen. Die Hook-Methoden lassen sich passend zu dem jeweiligen Kontext, in dem man sich befindet, überschreiben.

Die Java-Serialisierung bietet eine solche Hook-Methode an. Man implementiert einen speziellen *ObjectInputStream*, indem man von *ObjectInputStream* erbt und die passende Methode überschreibt:

```
public class PluginObjectInputStream extends ObjectInputStream {

  protected Class resolveClass(ObjectStreamClass desc)
      throws IOException, ClassNotFoundException {
    // spezielle Implementation zum Laden der gesuchten Klasse
  }

}
```

Natürlich muss die Methode so implementiert werden, dass sie die Klasse mithilfe des Classloaders des passenden Plugins lädt, indem die gesuchte Klasse definiert ist. Dazu müssen wir „nur“ noch das passende Plugin und den dazugehörigen Classloader identifizieren. Doch dazu später mehr.

Threads und *Context-ClassLoader*

Einige Bibliotheken nutzen für das dynamische Laden von Klassendefinitionen explizit den Classloader, der als *Context-ClassLoader* an dem aktuellen Thread gesetzt ist. Um einen solchen Classloader zu setzen, bietet die Klasse *java.lang.Thread* entsprechende Methoden an. Setzt man diesen Classloader nicht explizit, wird vom JDK implizit der Classloader des Parent-Threads gesetzt, was in der Regel der Classloader der Applikation ist.

Der Thread kann nun genutzt werden, um vor dem Aufruf einer Bibliothek den passenden Classloader zu setzen, der die von der Bibliothek dynamisch zu erzeugenden Klassendefinitionen laden kann. Allerdings sollte dies wirklich vor jedem Aufruf passieren, da der Code eines Plugins natürlich auch aus unterschiedlichen Threads heraus aufgerufen werden kann. Der passende Code im Client würde in etwa so aussehen:

```
public class Client {

  public void foo() {
```

```
    ClassLoader oldLoader =
        Thread.currentThread().getContextClassLoader();

    Thread.currentThread().setContextClassLoader(
        this.getClass().getClassLoader());

    myLibrary.foo();

    Thread.currentThread().setContextClassLoader(oldLoader);
  }

}
```

Für Bibliotheken ist es deshalb unter Umständen erforderlich, jeden Aufruf der Bibliothek durch eine entsprechende Wrapper-Klasse mit dem Setzen des *Context-ClassLoaders* zu kapseln. Im Client-Code muss dann der Wrapper verwendet werden anstelle der Original-Implementation. Dies kann allerdings bei einer großen API der Bibliothek schnell unhandlich werden. Zudem lauert das potentielle Problem, dass man nicht alle Aufrufe der API auf den Wrapper umleiten kann. Einige JDO-Implementationen beispielsweise generieren Code in zu persistierende Klassen, um spätes Laden zu realisieren. Da nicht mehr unbedingt vorherzusagen ist, wann dieser Code ausgeführt wird, kann ein entsprechender *Context-ClassLoader* nicht gezielt gesetzt werden.

Klassen gezielt aus Plugins laden

Nahezu alle bisher betrachteten Möglichkeiten setzen voraus, dass ein passender Classloader gefunden werden kann, der in der Lage ist, die gesuchten Klassendefinitionen zu laden.

Den Classloader zu einem Plugin zu ermitteln, ist in der Regel keine schwierige Aufgabe im Eclipse-Umfeld. Voraussetzung dafür ist, dass man ein Objekt oder eine Klasse des entsprechenden Plugins zur Hand hat. Der entsprechende Classloader lässt sich dann einfach abfragen:

```
ClassLoader pluginClassLoader =
  myObject.getClass().getClassLoader();
```

Hat man hingegen kein Objekt und keine Klasse des entsprechenden Plugins zur Hand, weiß aber, aus welchem Plugin die Klassen geladen werden soll, kann der Basis-OSGi-Mechanismus von Eclipse dazu verwendet werden. Dazu erfragt man zunächst das Bundle (ein Bundle ist das OSGi-Equivalent zu Eclipse Plugins. In Eclipse 3.0 werden alle Plugins

intern auf OSGi-Bundles abgebildet) des entsprechenden Plugins. An dem Bundle kann dann die passende Klasse mittels Bundle.loadClass(String className) geladen werden:

```
Bundle bundle = Platform.getBundle("my.own.plugin");
Class myClass = bundle.loadClass("my.own.plugin.MyClass");
```

Benötigt man gezielt den eigentlichen Classloader des Plugins, muss entweder ein spezieller Classloader als Wrapper um diese Zeilen implementiert werden. Alternativ kann man natürlich auch den Classloader der Klasse mit *myClass.getClassLoader()* direkt erfragen.

loadClass() != Class.forName()

Es kann Situationen geben, in denen das einfache Laden der Klasse mit *bundle.loadClass()* zu unschönen Seiteneffekten führt. Beispielsweise ist *bundle.loadClass()* nicht dazu geeignet, Array-Typen zu laden (was in der *resolveClass()*-Methode eines spezialisierten *PluginObjectInputStrems* unangenehm wäre). Außerdem ist die Klasse, nachdem sie über *bundle.loadClass()* geladen wurde, noch nicht initialisiert. Die Initialisierung führt die VM dann aus, wenn die Klasse das erste Mal verwendet wird. Dies kann speziell für *static*-initializer und *static*-Variablen zu unschönen Effekten führen.

Auf der sicheren Seite ist man stattdessen mit der *Class.forName()*-Operation, der man als zusätzlichen Parameter den Classloader mitgeben kann, mit dem die gesuchte Klasse geladen werden soll. Spätestens dann braucht man allerdings einen „echten" Classloader (und sei es nur der Wrapper um die *bundle.loadClass()*-Operation). Im Gegensatz zur *bundle.loadClass()*-Operation kann *Class.forName()* auch mit Array-Typen umgehen und sichert zu, dass die Klasse nach dem Laden sofort initialisiert wird.

UniversalPluginClassLoader

Möchte man Klassen aus allen möglichen Plugins des installierten Systems laden, lässt sich die *bundle.loadClass()*-Operation auch nutzen, um einen universellen Classloader zu implementieren. Dieser kann sich von einem *BundleContext* alle installierten Bundles des Systems geben lassen. Der universelle Classloader kann dann versuchen, die gesuchte Klasse nach dem Trial-and-Error-Verfahren von einem dieser Bundles laden zu lassen.

Dieses Verfahren hat natürlich den Nachteil, dass wirklich jedes Plugin nach der gesuchten Klasse gefragt wird und dadurch das Laden der Klassen sehr zeitaufwendig wird. Zudem ist nicht sichergestellt, dass wirklich die „richtige" Klasse geladen wird. Findet der universelle Classloader eine Klasse mit dem gleichen Namen zufällig in einem anderen Plugin, kann es passieren, dass eine falsche Klassendefinition geladen wird (mehr zum Finden der „richtigen" Klassendefinition s.u.).

Die richtige Klasse finden

Das Trial-and-Error-Verfahren kann bei einer Eclipse-basierten Plugin-Anwendung schnell an seine Grenzen geraten. Der Grund dafür liegt an der Eigenschaft der Eclipse-Runtime, die es erlaubt, dass mehrere Versionen eines Plugins gleichzeitig installiert sein können oder durch unterschiedliche Plugins Klassen gleichen Namens liefern können. Durch die Abhängigkeits-Definitionen zwischen den Plugins resultiert daraus in der Regel kein Problem für die Plugins selbst. Versucht man jedoch, eine Klassendefinition ausschließlich anhand ihres Namens zu finden, indem man alle installierten Plugins durchsucht, kann man schnell einmal das falsche Plugin (und damit die falsche Klassendefinition) in die Hände bekommen.

Sicherer ist es, wenn man exakt weiß, aus welchem Plugin die gesuchte Klasse geladen werden soll. Dies lässt sich bei der Java-Serialiserung beispielsweise dadurch realisieren, dass ein spezialisierter *ObjectOutputStream* beim Rausschreiben der Klasseninformationen zu den Objekten im Stream zusätzlich vermerkt, aus welchem Plugin die Klasse stammt. Ein spezialisierter *ObjectInputStream* kann diese Information auswerten und das entsprechende Plugin veranlassen, die Klasse zu laden. Diesen Weg kann man selbstverständlich nur dann einschlagen, wenn man sowohl die Serialisierung als auch die Deserialisierung unter Kontrolle hat. Wird die Serialisierung implizit (beispielsweise von RMI) genutzt, lassen sich meist keine spezialisierten *ObjectOutputStreams* einschleusen.

Das Trial-and-Error-Verfahren sollte deshalb nur in Notfällen und für solche Klassen eingesetzt werden, bei denen Mehrdeutigkeiten ausgeschlossen werden können.

Fazit

Dieser Artikel zeigt, dass das spezialisierte Classloading der Eclipse-Platform nicht nur Vorteile bringt. Zwar ist die klare Trennung von Plugins für mich ein klares Argument für die Eclipse-Platform. Allerdings muss man wissen, mit den kleinen Classloading-Tücken umzugehen. Ich hoffe, dieser Artikel kann dazu beitragen.

Abschließend danke ich Markus Völter für Feedback zu diesem Artikel sowie den Kollegen aus meinem aktuellen Projekt, die viele der hier beschriebenen Fallen und Lösungsideen mit mir gemeinsam durchlebt haben.

Autor

Martin Lippert ist langjähriger Consultant und Coach in den Bereichen agile Methoden, Software-Architekturen und Java. Er berät Entwicklungsorganisationen bei der Einführung und Anwendung agiler Entwicklungsmethoden und hilft Teams, neue Wege in der Softwareentwicklung zu bestreiten. Neben großen und komplexen Refactorings gehört die Entwicklung großer plugin-basierter Anwendungssysteme auf Basis der Eclipse-Plattform zu seinen Spezialgebieten. (lippert@acm.org, http://www.martinlippert.com/)