ToiletPaper #91

Java goes module

jambit WHERE INNOVATION WORKS

Autor: Arnim Kreutzer / Business Division Media

✖ Problem

Java hat sich seit seiner Veröffentlichung im März 1995 zu einer der populärsten und meistgenutzten Programmiersprachen entwickelt. Bei der Entwicklung spielten Anforderungen zur Beherrschung von Codebasen mit LOC-Zahlen im siebenstelligen Bereich bei der Konzeption keine Rolle. Von daher sind die implementierten Mechanismen zum Laden von Klassen - Stichwort Jar Hell - und der Zugriffssteuerung für solche Systeme unzureichend. Hierbei fehlt zwischen ***package*** und ***public*** ein Access Modifier, der den Zugriff auf Klassen aus bestimmten Packages einschränkt.
Unter anderem aus diesem Grund rief Sun 2008 das Projekt ***jigsaw*** mit dem Ziel ins Leben, die Entwicklung und Wartung großer Applikationen und Bibliotheken zu vereinfachen. Hierfür wurde ein Modulsystem konzipiert und in die Java-Plattform integriert.

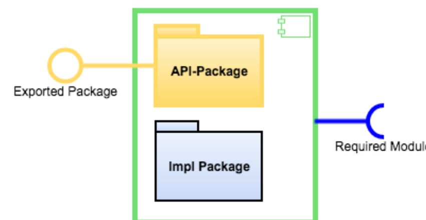
✔ Lösung

Mit Modulen werden Systeme nach fachlichen, technischen und/oder funktionalen Gesichtspunkten in abgeschlossene, lose gekoppelte Softwarekomponenten zerlegt. Auf der Java-Codeebene fassen Module Packages zu einer abgeschlossenen bzw. gekapselten Gruppe zusammen. Bei einem Modul muss – falls vorhanden – zwingend die API angegeben werden.

➔ Beispiel

Die Definition eines Moduls erfolgt über die Datei ***module-info.java.*** Diese muss im root-Verzeichnis des Sourcetrees liegen und sieht in einem einfachen Fall wie folgt aus:

```
module my.module.name {
    exports my.exported.api.pckg;
    requires name.of.required.module;
}
```



Ein Modul kann mehrere Packages exportieren und (natürlich) von mehreren Modulen abhängig sein. Für jedes exportierte Package bzw. jedes benötigte Modul wird eine Zeile mit einer exports bzw. requires Direktive eingetragen. Das API eines Moduls besteht aus sämtlichen public-Elementen der exportierten Packages.
Bereits zur Compilezeit überprüft das Modulsystem die Abhängigkeiten und erstellt ausgehend von einem Root-Modul den Modulgraph, in dem Knoten die Module und Kanten die Abhängigkeiten darstellen. Zyklische Abhängigkeiten zwischen Modulen sind nicht erlaubt und führen bei Erkennung zu einem Fehler. Der Modulgraph ist demnach garantiert azyklisch.
Durch die Verankerung der oben genannten Eigenschaften in der JVM ergeben sich die folgenden Vorteile des Java Platform Module Systems gegenüber den vorherigen JVM-Versionen:

1. Verlässliche Konfiguration: Beim Start der JVM werden die Abhängigkeiten geprüft. So ist zu diesem Zeitpunkt bereits sicher, dass alle erforderlichen Module vorhanden sind.

2. Strenge Kapselung: Durch die Kapselung wird verhindert, dass auf interne Klassen – die zwecks package-übergreifender (modulinternen) Nutzung public sein müssen – von außerhalb des Moduls zugegriffen werden kann.

3. Skalierbare Entwicklung: Module können in eigenständigen Teams entwickelt werden. Die Abgrenzung der Module erfolgt automatisch durch das Modulsystem über den Austausch der explizit exportierten Teile.

4. Sicherheit: Die Kapselung ist auf der untersten Ebene der JVM realisiert. Der Zugriff über Reflection auf sensible interne Klassen ist nicht mehr möglich.

5. Optimierungen: Über den Modulgraph ist festgelegt, welche Module inkl. der Plattformmodule für die Ausführung der Applikation benötigt werden. Beim Start der JVM werden ausschließlich diese Module berücksichtigt. Des Weiteren kann für die Distribution eine minimale Konfiguration erzeugt werden. Das immer wieder notwendige lineare Durchsuchen des Classpath beim Laden von Klassen entfällt, da die Kombination Modul/ Package systemweit eindeutig ist.

✚ Weiterführende Aspekte

- Implementierung von Services
- Module testen
- Migration & Automated modules
- Reflection vs. Strong encapsulation
- Module Descriptor advanced
- Modulelayer/ Configuration
- API-Design
- Runtime-Images mit jlink (JEP-220, JEP-282)
- jamb.it/coffeetalks-2018-3