Fachhochschule
Münster University of
Applied Sciences

Das Java Modulsystem

---

fast neue

Projekt ***Jigsaw (JSR 376)***: das ~~neue~~ Modulsystem des JDK1.9+

Prof. Dr. Nikolaus Wulff

Project Jigsaw

Das Projekt Jigsaw (*engl. Laubsäge*) bildet seit dem JDK 1.9 (Sept. 2017) das neue Java Modulsystem.

- Es erlaubt massgeschneiderte Abhängigkeiten und Sichtbarkeiten zwischen JAR Archiven, die als Module gepackt werden (können).
- Es verbietet zyklische Abhängigkeiten zwischen Modulen (zur Compile-Zeit) und erzwingt eine saubere Schichten- & Modularchitektur.
- Es erlaubt Interna – d.h. Pakete und Klassen –, von Modulen vor der Außenwelt zu verschließen.
- Es hat nichts mit *Maven* & *Gradle* Modulen zu tun.

Sichtbarkeiten (*access-modifier*)

- Java kennt für Methoden und Attribute die vier Sichtbarkeiten ***public***, ***protected***, ***private*** und ***default*** (***package***) auf Klassenebene. (=> *Wiederholung der 4p Info II*)
- Für Klassen gibt es lediglich die zwei Sichtbarkeiten ***public*** und ***default*** (***package***) auf der Paketebene.
- Die Sichtbarkeit von Paketen konnte bis JDK1.9 nicht eingeschränkt oder geregelt werden.
  - Firmen wählten z.B. Namen wie: *com.xyz.internal.util*, um Pakete als interne private API zu kennzeichnen ...
  - Klassen wurden dann häufig mit der *default* Sichtbarkeit *package-protected* vor der Außenwelt versteckt.
  - All dies war unvollkommen, anfällig und kritikwürdig ...

Sichtbarkeiten cont.

- Per Reflection API war es möglich auch auf private & protected Attribute oder Methoden zuzugreifen.
  - Tools wir *Spring* oder *Hibernate* nutzten dies z.B. zur Instanziierung nach dem DIP/IoC Prinzip per *@Inject*.
- Es war möglich, dass eine Klasse „behauptete“ in einem Paket einer externen JAR zu sein, um auf dessen Interna zuzugreifen, indem ein Entwickler ein gleichnamiges Paket in seinem Projekt deklarierte. Dies führte zu sogenannten „split-packages“. Seit dem JDK1.9 ist dies mit Jigsaw nicht mehr möglich.
- Viele Firmen blieben genau deshalb beim JDK1.8 bzw. verwenden 1.9 – 1.15 mit compile-option 1.8!

module-info.java

- Seit dem JDK1.5 existiert die Datei *package-info.java* sie hat keine weiteren Auswirkungen, enthält keinen Code und dient lediglich zur Dokumentation.
- Mit dem JDK1.9 wurde die Datei *module-info.java* eingeführt, die zur Spezifikation eines Moduls dient.
  - Diese Datei muss sich im sonst verpönten top-level *default* Paket (d.h. Paket ohne Namen) befinden.
  - In dieser Datei werden der Name des Moduls, alle Abhängigkeiten, alle Exporte nach Außen und weitere Zugriffsmöglichkeiten wie z. B. Reflection geregelt.
  - Sobald in einem Projekt eine Modulbeschreibung existiert müssen alle damit verbundenen Regeln eingehalten werden.

Beispiel einer module-info

```
module scripting {
    requires transitive core;
    requires transitive java.scripting;
    requires antlr4.runtime;

    exports de.lab4inf.wrb2.scripting;

    uses javax.script.ScriptEngineManager;

    opens de.lab4inf.wrb2.scripting to javax.script;

    provides javax.script.ScriptEngineFactory with
                de.lab4inf.wrb2.scripting.WRBScriptEngineFactory;

}
```

Import von Anderen

Export an Dritte

Reflection für javax

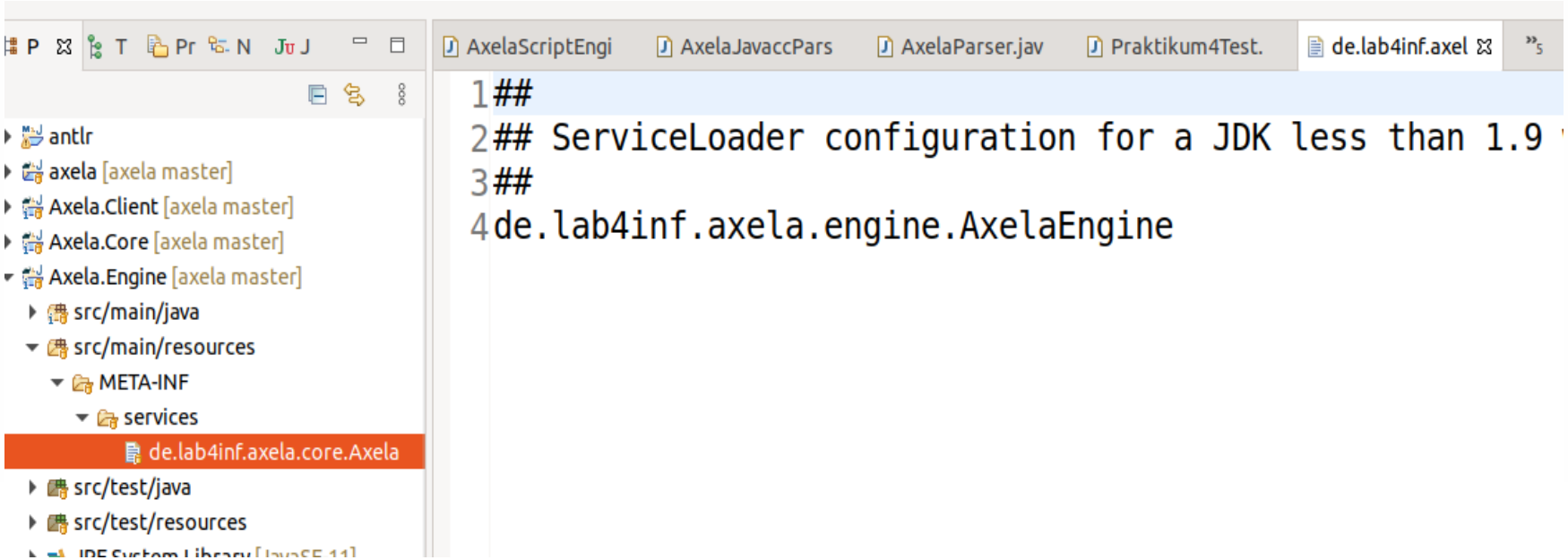
Implementierung für javax ...

- Alle In- und Exporte werden explizit deklariert.
- Reflection für alle oder bestimmte Pakete muss geöffnet werden.

JDK 1.8 versus 1.9

- Service Implementierungen wurden bis zum JDK 1.8 per **ServiceLoader<T>** ermittelt, indem im META-INF/service Verzeichnis eine entsprechende Datei angelegt wurde.
- Seid JDK 1.9 wird dies mit einem „provides“ Eintrag in der module-info hinterlegt. Die Implementierung benötigt eine statische provides Methode.
- Probleme entstehen, wenn alte JAR Dateien verwendet werden, die nicht dem Modulsystem genügen. Diese werden als „Automatic Module“ gelesen und mit allen anderen JAR Dateien transitiv auf dem Classpath exportiert und für Reflection geöffnet.

Axela per Reflection



- Bis JDK1.8 wurden Implementierungen mittels einer Datei im META-INF/services Verzeichnis deklariert.
- Seid dem JDK1.9 steht dies per „provides“ in der Modulbeschreibung.

Zusammenfassung

- Das JDK1.9 erlaubt eine klare, feingranulare Modulstruktur mit expliziten Zugriffsrechten.
- Zyklische Abhängigkeiten und „split-packages“ sind nicht mehr möglich, was eine saubere Architektur unterstützt.
- Paketen (und Klassen) können nach Außen versteckt oder gezielt für „alle“ oder bestimmte Module geöffnet werden.
- Die Umstellung einer „Altanwendung“ ist aufwendig aber machbar und erfordert unter Umständen Refactoring der Abhängigkeiten.