



Workshop

Modularisierung mit

Project Jigsaw

Michael Inden

Freiberuflicher Consultant, Buchautor und Trainer

Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre TPL / SA bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre LSA / Trainer bei Zühlke Engineering AG in Zürich
- ~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant und Trainer**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch

Blog: <https://jaxenter.de/author/minden>

Kurse: Bitte sprecht mich an!





Agenda

Workshop Contents



- **Modularisierung mit Project Jigsaw**
 - PART 1: Einführung
 - PART 2: Sichtbarkeiten und transitive Abhängigkeiten
 - PART 3: Abhängigkeiten mit Services lösen
 - PART 4: Externe Module einbinden / Migrationen
 - **Fazit**
-



PART 1: Einführung



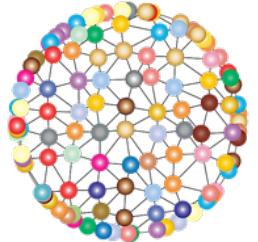
Einführung: Ziele der Modularisierung



- **Modularisierung von Anwendungen und Bibliotheken ermöglichen**
 - **Modularisierung des JDKs an sich**
 - **Abhängigkeiten und Sichtbarkeiten steuern**
 - **zuverlässige Konfiguration statt fehlerträchtiger Abhängigkeitsverwaltung**
-



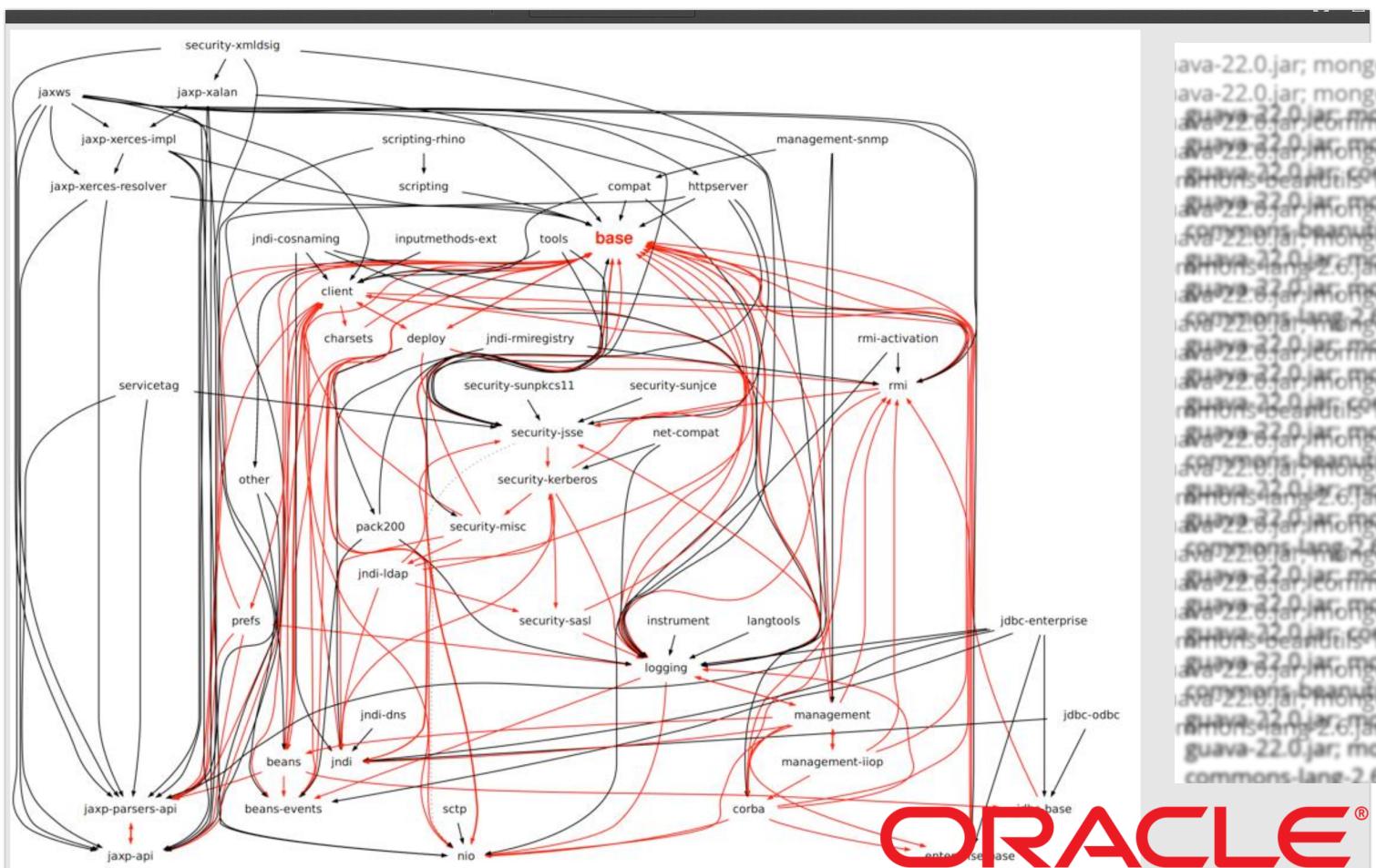
**Aber warum ist das
erstrebenswert?**



Einführung: Gründe für Modularisierung



- Situation mit Java 8: **Wirre Abhangigkeiten** und **Chaos** im CLASSPATH



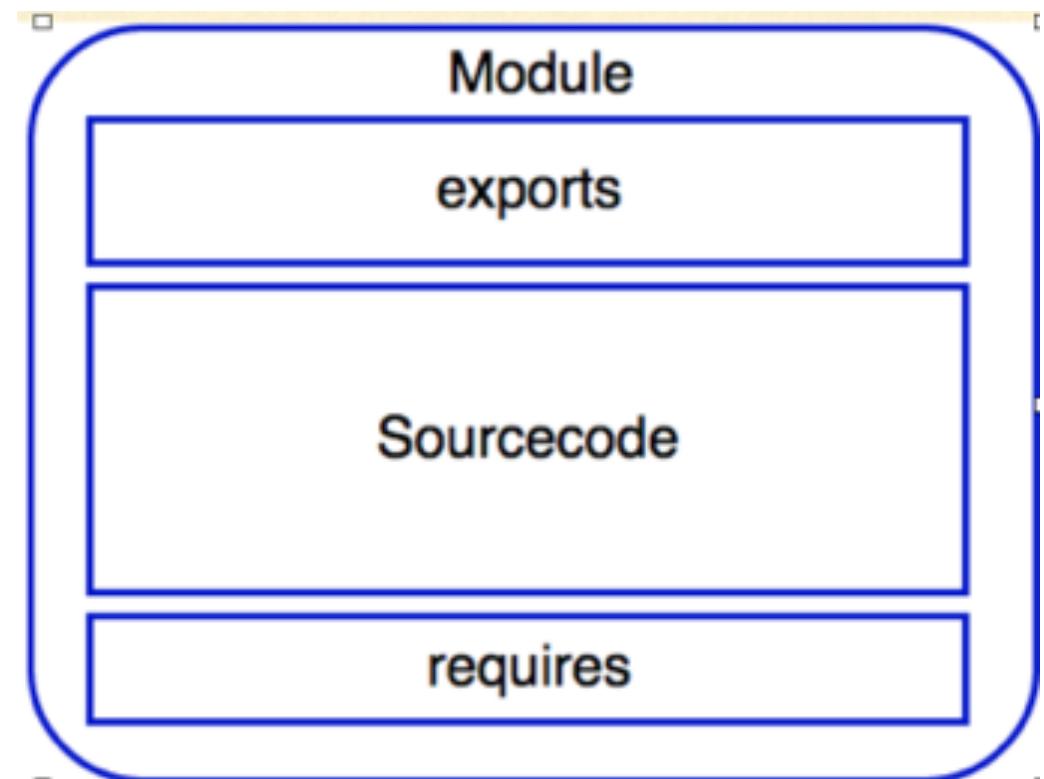


Situation mit Java 8:

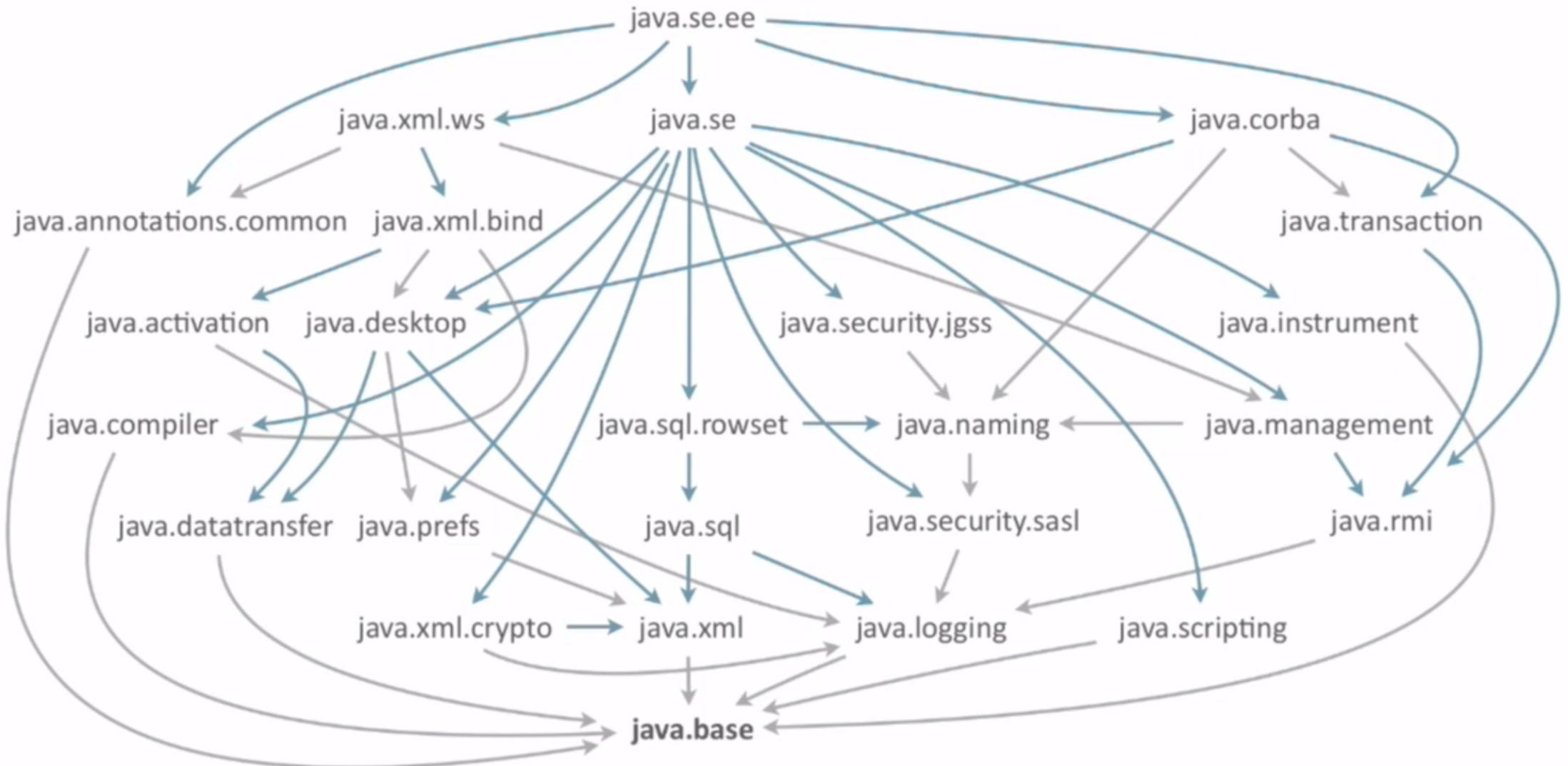
- „**Module**“ als Sammlung von Klassen in Packages bzw. als JARs
 - lose Kopplung lässt sich nicht forcieren
 - Abhängigkeiten und Sichtbarkeiten schwierig zu steuern
- „**Module**“ mithilfe von OSGi
 - sehr ausgereift
 - aber etwas komplex und nicht für das JDK geeignet



- **Module als neue Softwarebausteine im JDK als Gruppierung von Packages**
- **Ein Modul ...**
 - **besitzt einen eindeutigen Namen**
 - **besteht aus Packages, Klassen usw.**
 - **Bietet abgegrenzte Funktionalität**
 - **versteckt Implementierungsdetails**
 - **definiert sämtliche Abhängigkeiten**
 - **besitzt wohldefinierte Schnittstelle**



Einführung: Modularisierung des JDKs

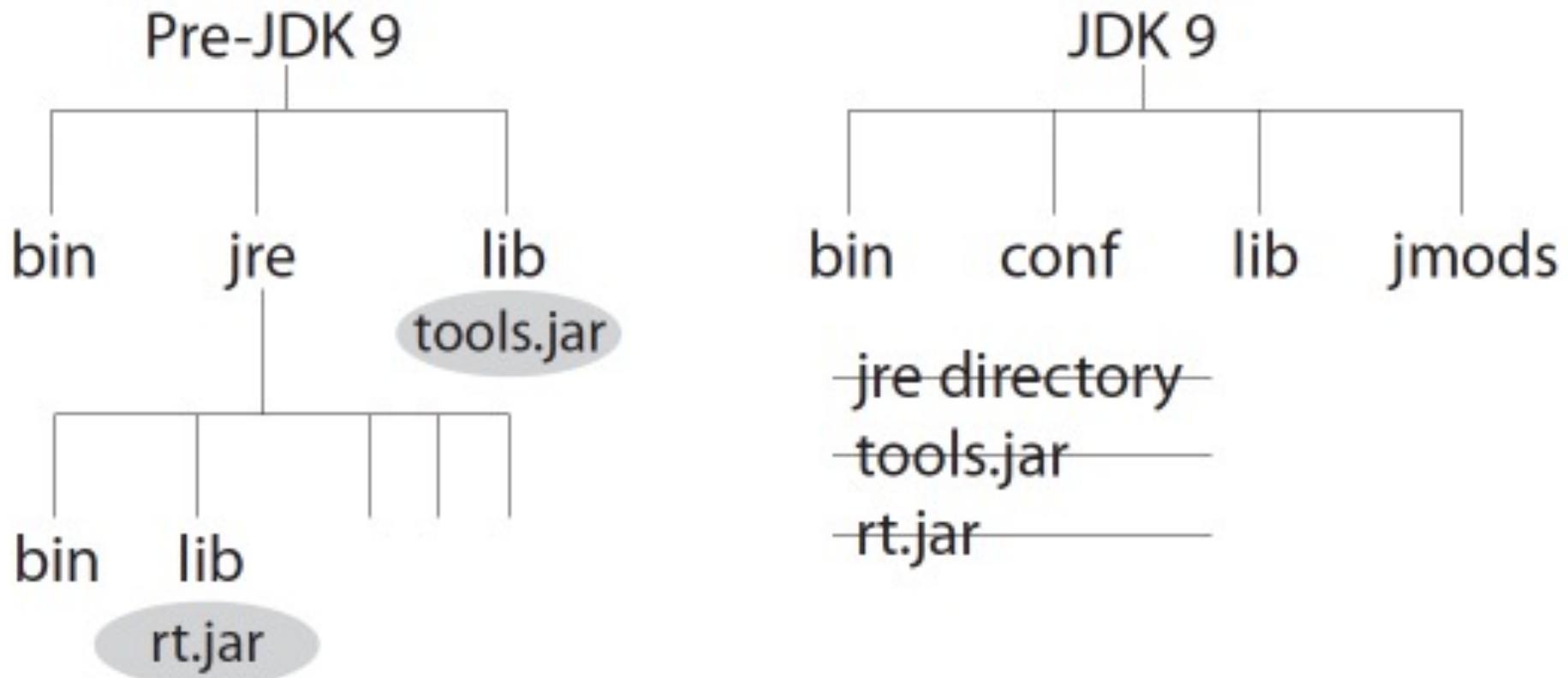


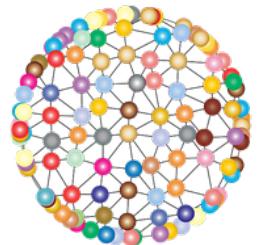
Einführung: Modularisierung des JDKs



- **JDK-Verzeichnisstruktur massiv geändert, kein JDK / JRE mehr**
 - **rt.jar und tools.jar existieren nicht mehr => Module in jmods**
 - **Sichtbarkeitsbeschränkungen => einige interne APIs nicht mehr zugreifbar**
 - **Erweiterung des JDKs mit »endorsed dirs« nicht mehr unterstützt**
 - **Split Packages nicht mehr unterstützt**
 - **Tool jlink, mit dem man spezielle Executables des eigenen Programms erstellen kann**
-

Einführung: Modularisierung des JDKs





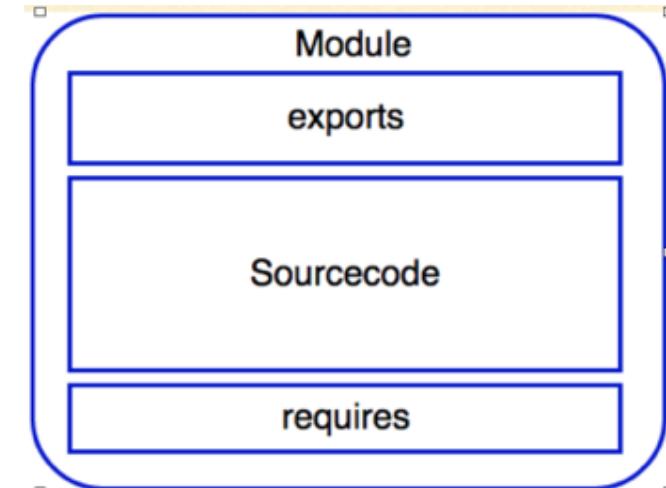
**Und wie sehen jetzt
Module aus?**

Einführung: Module in Java 9



- **Module als neue Softwarebausteine im JDK als Gruppierung von Packages**
- **Jedes Modul besitzt einen Moduldeskriptor module-info.java**

```
module <ModuleName>
{
    requires <ModuleNameOfRequiredModule>;
    exports <PackageName>;
}
```



- **Enthält eine Menge von Packages und Klassen**
- **Definiert die Menge an Abhängigkeiten**
- **Das Modulsystem stellt sicher, dass jede Abhängigkeit genau durch ein Modul erfüllt wird und die Abhängigkeiten azyklisch sind.**

Beispiel 2 Module: Kompilieren, Anpassung im Java-Ökosystem



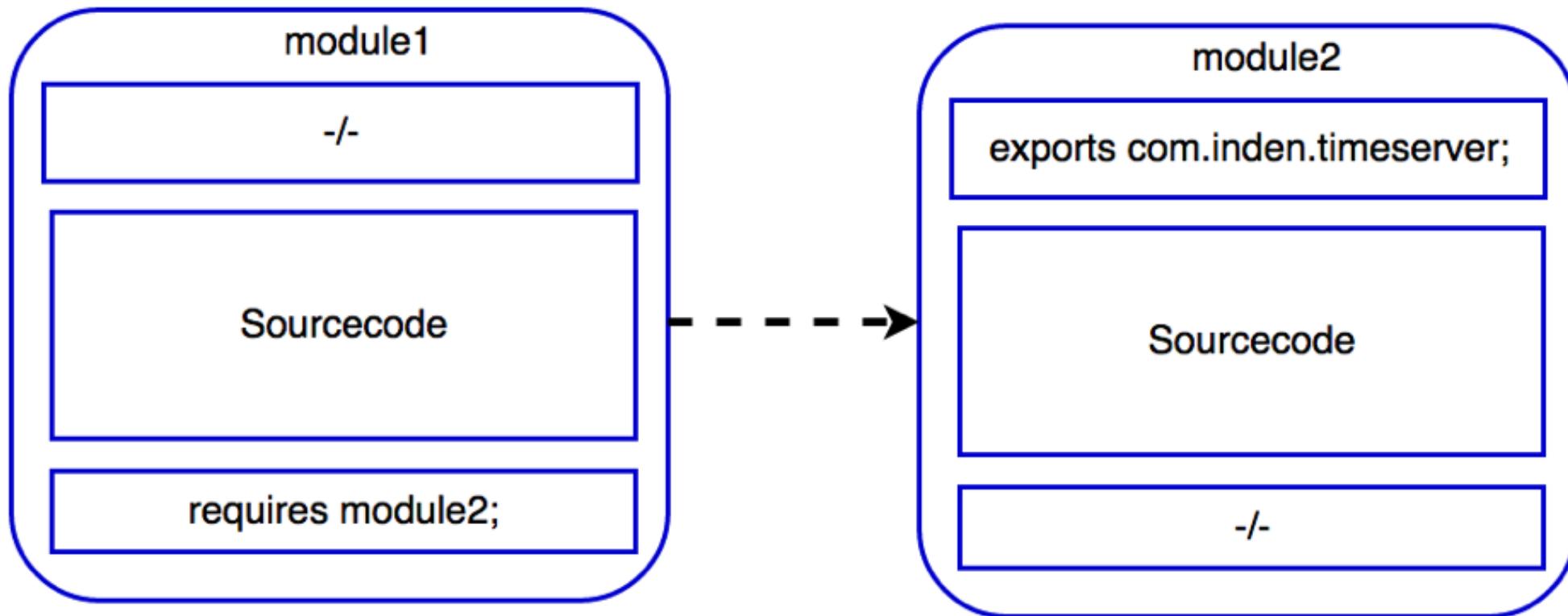
- **Module stellen eine weitere Hierarchie zu Packages dar**
- **Daher wurden Erweiterungen nötig**
- **Java-Compiler wurde um Module-Path-Angabe erweitert:**

```
javac --module-path <module-path> ... oder javac -p <module-path> ...
```

- **Java-Runtime wurde um Module-Path und zu startende Klasse mit Module erweitert:**

```
java --module-path <modulopath> -m <modulename>/<moduleclass>
```

Einführung: Module in Java 9



```
module module1
{
    requires module2;
}
```

```
module module2
{
    exports com.inden.timeserver;
}
```



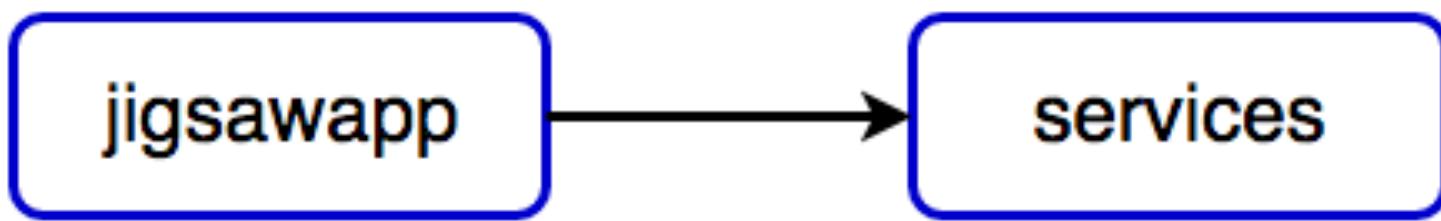
- **Modul 1 requires Modul 2 => Modul 1 reads Modul 2**
- **Readability** ist die Voraussetzung dafür, dass Modul 1 die Typen aus Modul 2 referenzieren kann.
- **Accessibility:** Readability + exports
Eine Klasse A aus Modul 2 ist nur dann zugreifbar, wenn Modul 1 das korrespondierende Modul 2 liest und Modul 2 zusätzlich das benötigte Package exportiert. => starke Kapselung
- Beides bildet die Grundlage für eine **verlässliche Konfiguration**.



Beispiel mit 2 Modulen



Beispiel 2 Module



Verzeichnislayout (Oracle)



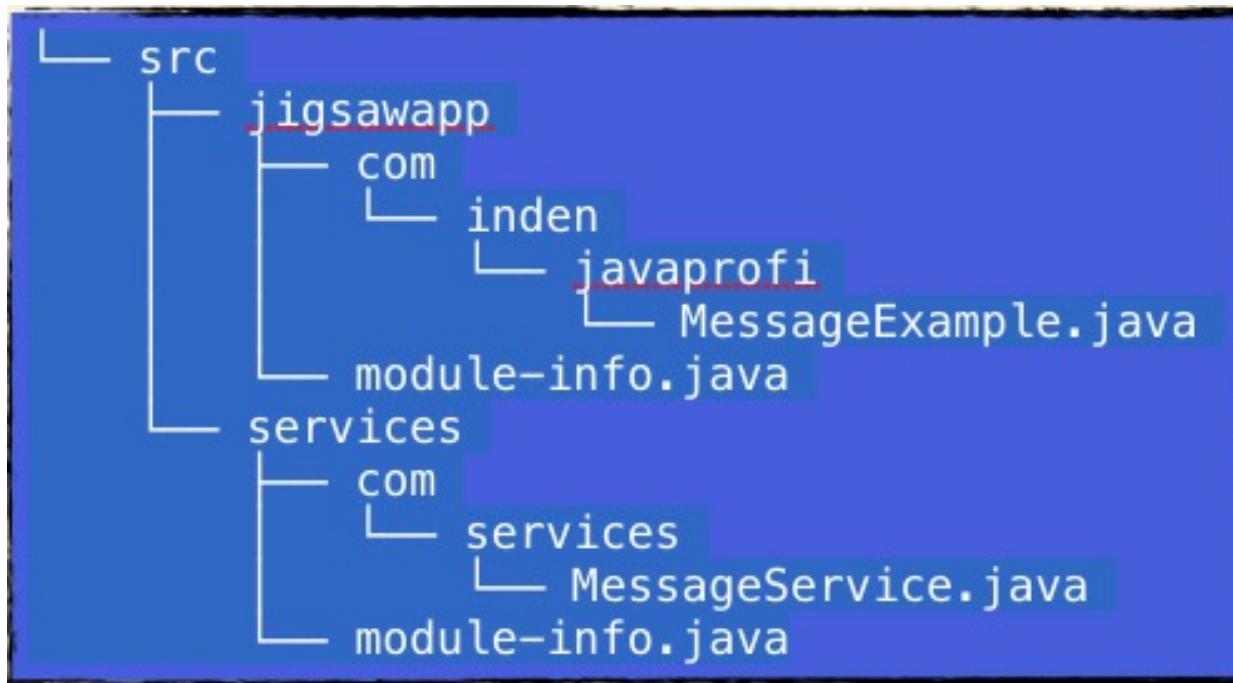
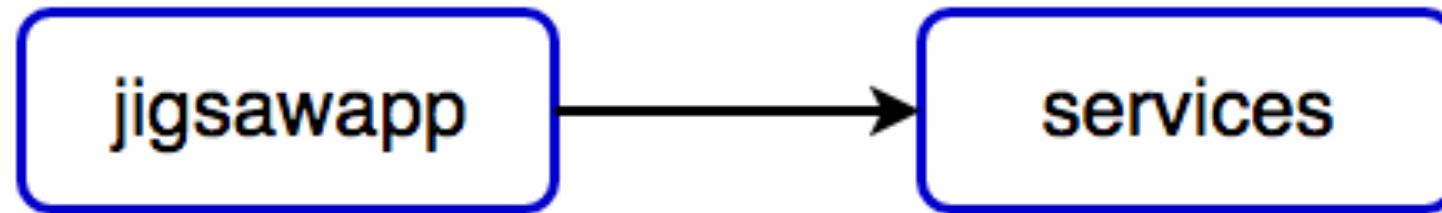
- Applikation mit mehreren Modulen => common src-Verzeichnis
- Pro Modul: Sourcecode in einem Unterverzeichnis mit dem Modulnamen abgelegt

```
'-- src
  |-- com.inden.module1
  |   |-- com
  |   |   '-- inden
  |   |       '-- module1
  |   |           '-- Application.java
  |   '-- module-info.java
  '-- com.inden.module2
      |-- com
      |   '-- inden
      |       '-- module2
      |           '-- OtherClass.java
      '-- module-info.java
```

- Nur für spezielle Anwendungsfälle und erste Beispiele praktisch



Beispiel 2 Module: Abhangigkeit und Verzeichnisstruktur



Beispiel 2 Module: Verzeichnisstruktur + Implementierung



- **Moduldeskriptoren**

```
module jigsawapp {  
    requires services;  
}
```

```
module services {  
    exports com.services;  
}
```

- **Implementierung der Klasse für das Modul services**

```
package com.services;  
  
public class MessageService  
{  
    public String createGreetingMessage(final String name)  
    {  
        return "Hello " + name;  
    }  
}
```

Beispiel 2 Module: Verzeichnisstruktur + Implementierung



- **Implementierung der Klasse des Moduls jigsawapp**

```
package com.inden.javaprofi;  
  
import com.services.MessageService;  
  
public class MessageExample  
{  
    public static void main(String[] args)  
    {  
        var msgService = new MessageService();  
        System.out.println(msgService.createGreetingMessage("Mainz"));  
    }  
}
```

Beispiel 2 Module: Kompilieren



- **Kompilieren erst Modul services, dann Modul jigsawapp**

```
javac -d build/services \
      src/services/*.java \
      src/services/com/services/*.java
```

```
javac -d build/jigsawapp \
      src/jigsawapp/*.java \
      src/jigsawapp/com/inden/javaprofi/*.java
```

```
=> src/jigsawapp/module-info.java:2: error: module not found: services
    requires ^

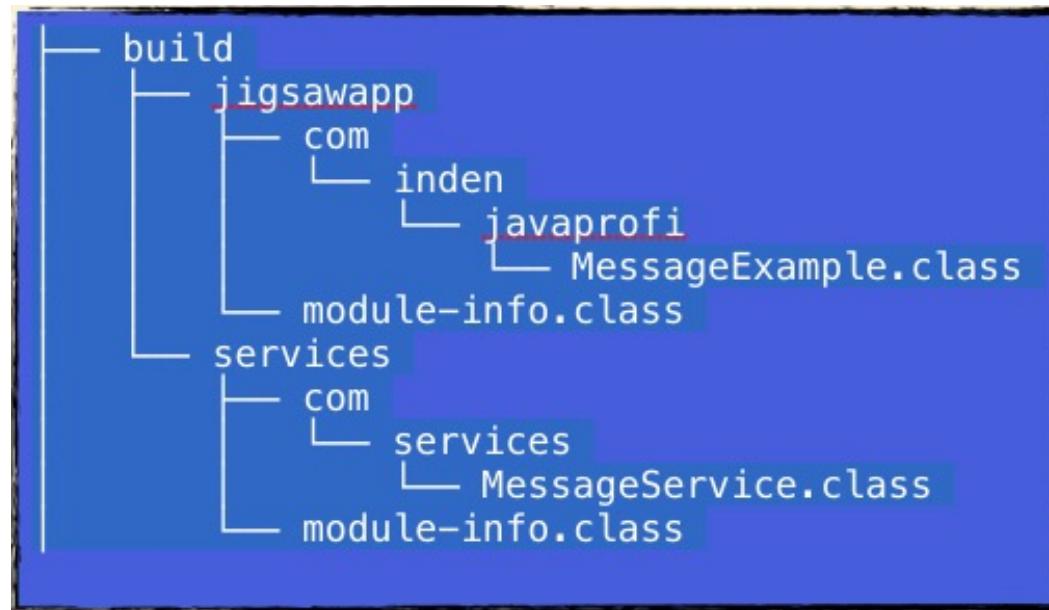
1 error
```

Beispiel 2 Module: Kompilieren



- **Kompilieren des Moduls jigsawapp mit Module-Path**

```
javac -d build/jigsawapp \
      -p build \
      src/jigsawapp/*.java \
      src/jigsawapp/com/inden/javaprofi/*.java
```



Beispiel 2 Module: Packaging und Applikationsstart

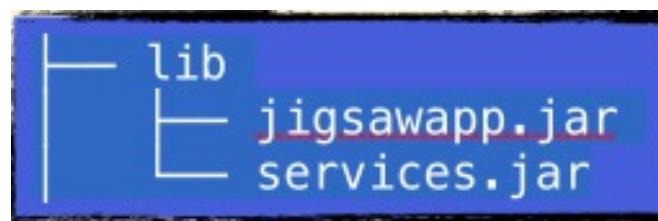


- **Packaging**

```
mkdir lib
```

```
jar --create --file lib/services.jar -C build/services .
```

```
jar --create --file lib/jigsawapp.jar -C build/jigsawapp .
```



- **Applikationsstart**

```
java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample
```

Beispiel 2 Module: Packaging und Applikationsstart

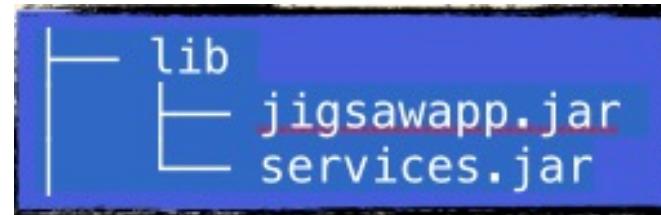


- **Packaging**

```
mkdir lib
```

```
jar --create --file lib/services.jar -C build/services
```

```
jar --create --file lib/jigsawapp.jar \  
--main-class com.inden.javaprofi.MessageExample \  
-C build/jigsawapp .
```

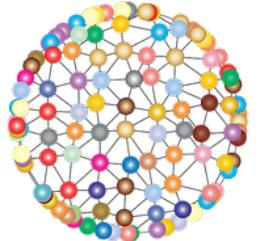


- **Applikationsstart**

```
java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample  
java -p lib -m jigsawapp
```



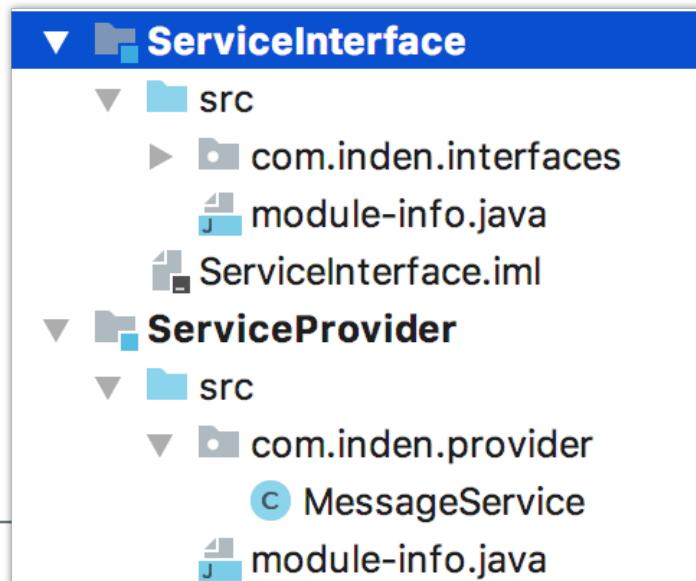
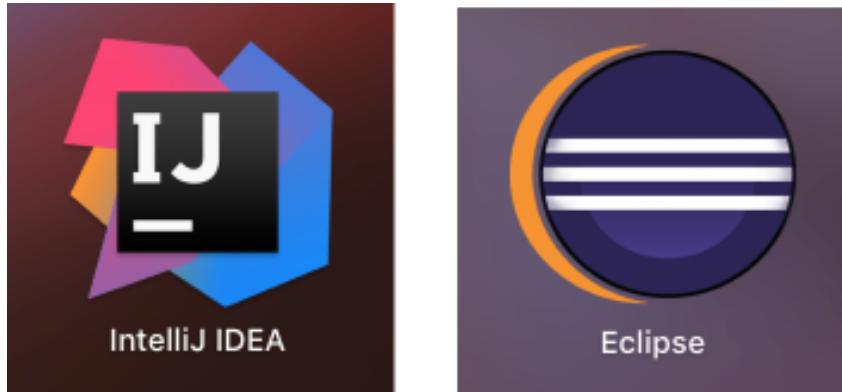
Module als High-Level-Bausteine, dann aber Low-Level-Konsolen-Junkie?



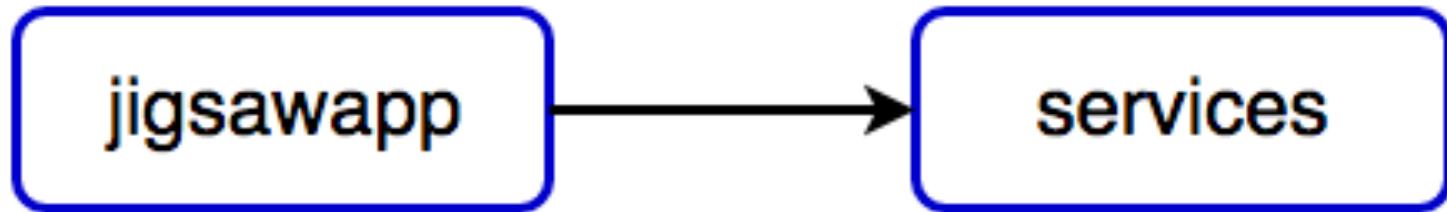
IDE Support



- Aktuelle IDEs grundsätzlich gut
- Eclipse: Module entsprechen Projekten
- IntelliJ: Spezielles Modulkonstrukt unterhalb von Projekten
- Noch kein Standardlayout, diverse Variationen möglich
- Nutze gerne das normale Layout ohne den Modulnamen im «src»



Beispiel 2 Module: Verzeichnisstruktur in der IDE



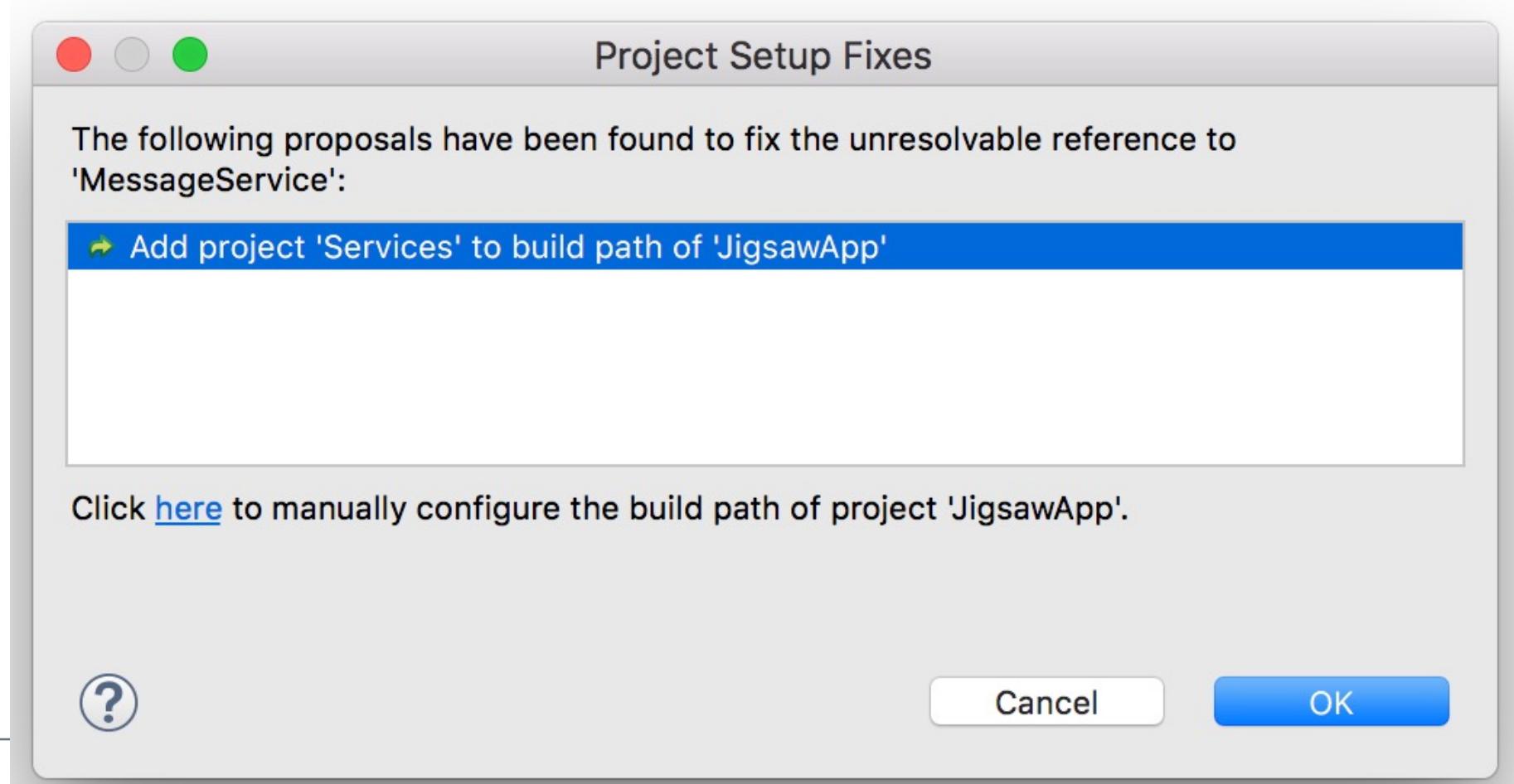
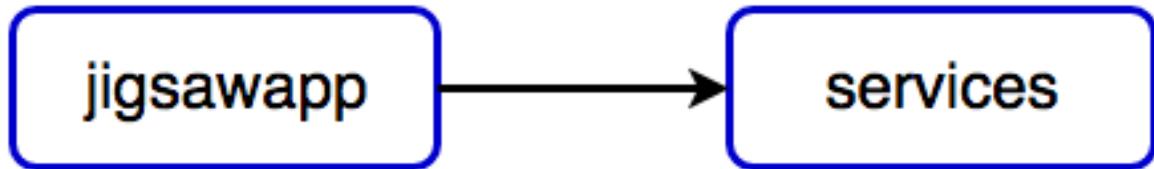
JigsawApp

- JRE System Library [Java12]
- ▼ src
 - ▼ com.inden.javaprofi
 - MessageExample.java
 - ▼ module-info.java
 - jigsawapp

Services

- JRE System Library [Java12]
- ▼ src
 - ▼ com.services
 - MessageService.java
 - ▼ module-info.java
 - services

Beispiel 2 Module: Abhangigkeit in der IDE



Beispiel 2 Module: Packaging und Applikationsstart



- **Das war es schon!** Viel besser als auf der Konsole, oder? **
- Zwischenfazit
 - **Saubere Struktur**
 - **Separate Einheiten als Projekte / Module**
 - **Separates Bauen und Weiterentwickeln möglich**
 - **Lässt sich leicht auf die von Build-Tools erwartete Struktur bringen**
 - **Bessere Verzeichnisstruktur: Benötigt kein künstliches Zwischenverzeichnis**
 - **Separate Bereitstellung als JAR einfach möglich**



Besonderheiten



Abhängigkeiten auflisten



- Ermitteln von Abhängigen: `jdeps lib/*.jar`

```
jigsawapp
[file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch6/
    ch6_2_6_dependencies_module_example/lib/jigsawapp.jar]
    requires mandated java.base (@9-ea)
    requires services
jigsawapp -> java.base
jigsawapp -> services
    com.inden.javaprofi      -> com.services.api          services
    com.inden.javaprofi      -> java.io                  java.base
    com.inden.javaprofi      -> java.lang                java.base
    com.inden.javaprofi      -> java.lang.invoke        java.base
services
[file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch6/
    ch6_2_6_dependencies_module_example/lib/services.jar]
    requires mandated java.base (@9-ea)
services -> java.base
    com.services.api          -> com.services.impl        services
    com.services.api          -> java.lang                java.base
    com.services.impl         -> com.services.api        services
    com.services.impl         -> java.lang                java.base
```

- **HINWEIS:** Für IDE-Variante müssen wir lediglich die jeweiligen JARs in gemeinsames lib Verzeichnis kopieren, dann können wir die Aktionen gleich ausführen!

Abhängigkeiten kompakt auflisten



- **Kompakte Aufbereitung der Abhängigkeiten**

```
jdeps -s lib/*.jar
```

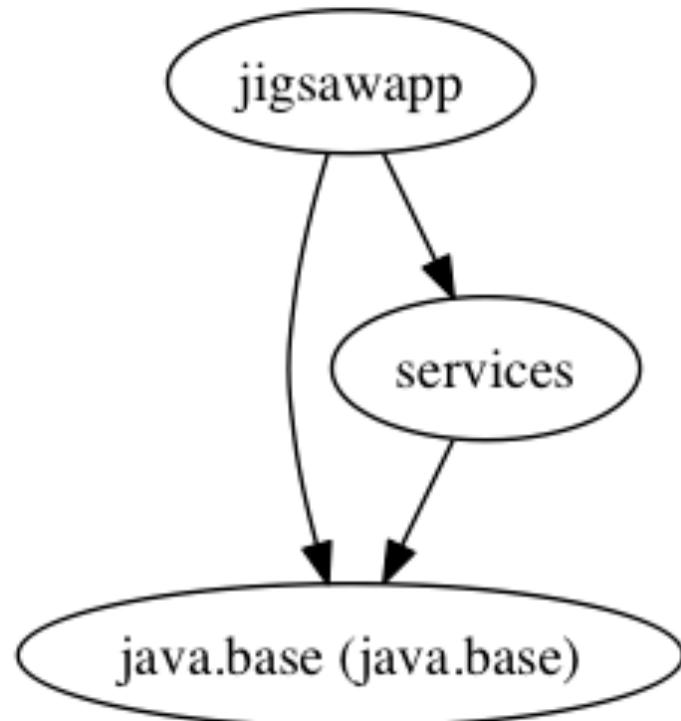
```
jigsawapp -> java.base  
jigsawapp -> services  
services -> java.base
```

Abhängigkeitsgraph erstellen



- Aufbereitung eines Abhängigkeitsgraphen

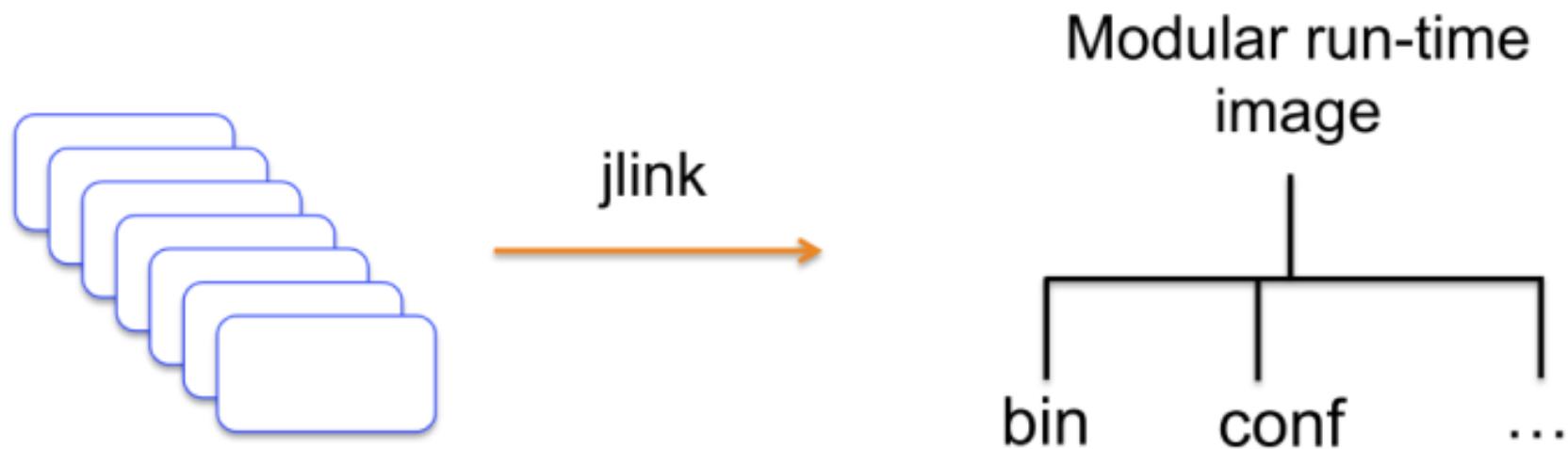
```
jdeps --module-path build -dotoutput graphs lib/*.jar
```





- **Executable mit inkludierter Java Runtime erstellen**

```
jlink --module-path $JAVA_HOME/jmods:lib --add-modules jigsawapp \
--launcher jigsawapp=jigsawapp/com.inden.javaprofi.MessageExample \
--output exec_example
```





```
'-- exec_example
  |-- bin
  |   |-- java
  |   |-- jigsawapp
  |   '-- keytool
  |-- conf
  |   |-- net.properties
  |   '-- security
  |       |-- java.policy
  |       |-- java.security
  |       '-- policy
  |           |-- README.txt
```

Start der Applikation

Zum Starten des Programms gibt man Folgendes ein:

```
./exec_example/bin/jigsawapp
```



JDK-Module einbinden

Module des JDKs einbinden

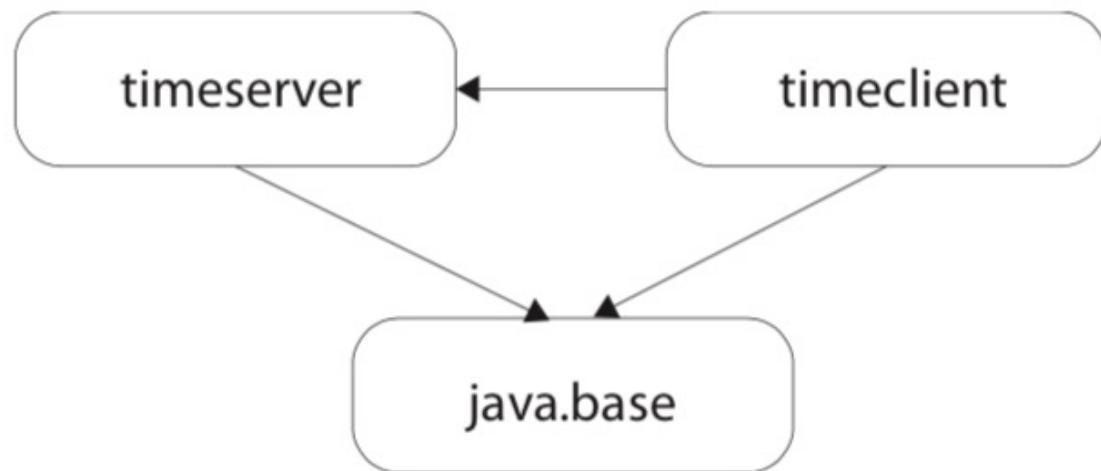
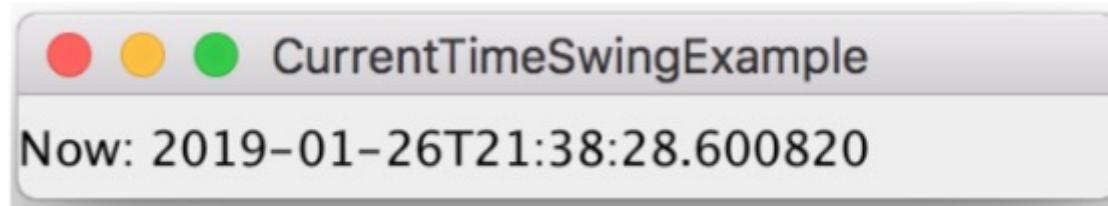


- Bislang nur eigene Funktionalität genutzt => **eher unrealistisch**
- Was ist, wenn wir Dinge aus dem JDK benötigen?
 - => einiges schon im Modul `java.base`, die Basis aller Module analog zur Klasse `Object`
 - => **Module des JDKs müssen explizit im eigenen Moduldeskriptor aufgeführt werden**

Module des JDKs einbinden



Beispiel: modularisierte Swing-Applikation, die in einem Fenster die aktuelle Zeit anzeigt



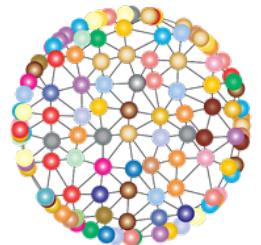
Module des JDKs einbinden



Beispiel: modularisierte Swing-Applikation, die in einem Fenster die aktuelle Zeit anzeigt

```
module-info.java build.gradle *HelloJigsaw.java Classresolver.java ReflectionExample

1 package com.hellojigsaw;
2
3 import javax.swing.JFrame;
4 import javax.swing.JLabel;
5
6 public class CurrentTimeSwingExample
7 {
8     public static void main(final String[] args)
9     {
10         final JFrame appFrame = new JFrame("CurrentTimeSwingExample");
11         final JLabel label = new JLabel("Now: " + TimeInfo.getCurrentTime());
12
13         appFrame.getContentPane().add(label);
14
15         appFrame.setSize(300, 50);
16         appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         appFrame.show();
18     }
19 }
```



**Woher wissen wir, welche
Module wir dazu einbinden
müssen?**

Hinweise durch den Compiler oder die IDE



```
src/timeserver/com/server/TimeInfo.java:4: error: package java.util.logging is  
    not visible  
import java.util.logging.Level;  
^  
  
(package java.util.logging is declared in module java.logg  
    timeserver does not read it)  
src/timeserver/com/server/TimeInfo.java:5: error: package ja  
    not visible  
import java.util.logging.Logger;  
^  
  
(package java.util.logging is declared in module java.logg  
    timeserver does not read it)  
src/timeclient/com/client/CurrentTimeSwingExample.java:5: er  
    swing is not visible  
import javax.swing.JFrame;  
^  
  
(package javax.swing is declared in module java.desktop, b  
    does not read it)  
src/timeclient/com/client/CurrentTimeSwingExample.java:6: er  
    swing is not visible  
import javax.swing.JLabel;  
^  
  
(package javax.swing is declared in module java.desktop, b  
    does not read it)  
src/timeclient/com/client/CurrentTimeSwingExample.java:24: e  
    symbol  
    appFraem.pack();  
^  
  
symbol:   variable appFraem  
location: class CurrentTimeSwingExample  
5 errors
```

final JFrame appFrame = new JFrame("CurrentTimeSw:
final JLabel label = new JLabel("Now: " + TimeInfo

appFra



JLabel cannot be resolved to a type

6 quick fixes available:

- C [Create class 'JLabel'](#)
- ⚡ [Add 'requires java.desktop' to module-info.java](#)
- I [Create interface 'JLabel'](#)
- E [Create enum 'JLabel'](#)
- ⌚ [Add type parameter 'JLabel' to 'main\(String\[\]\)'](#)
- ⚡ [Fix project setup...](#)



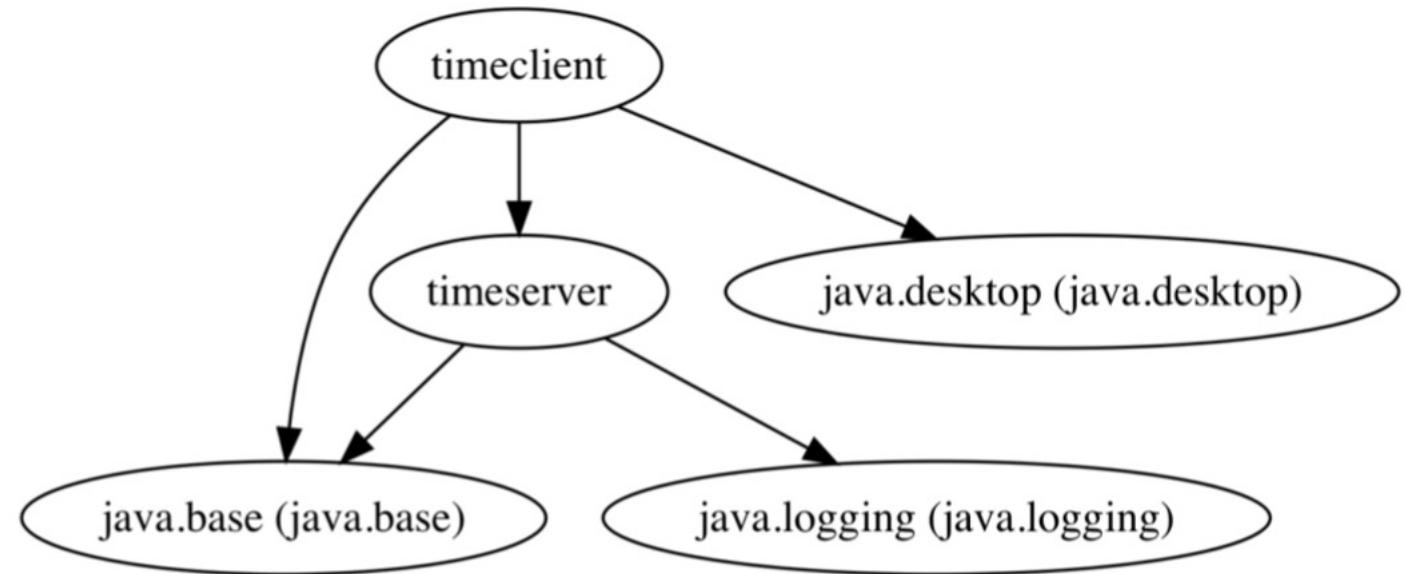
Press 'F2' for focus

Module des JDKs einbinden: Moduldeskriptoren anpassen



```
module timeclient
{
    requires java.desktop;
    requires timeserver;
}
```

```
module timeserver
{
    requires java.logging;
    exports com.server;
}
```





Anmerkungen zum Verzeichnislayout

Recap Verzeichnislayout (Oracle)



- Applikation mit mehreren Modulen => gemeinsames src-Verzeichnis
- Pro Modul: Sourcecode in einem Unterverzeichnis mit dem Modulnamen abgelegt

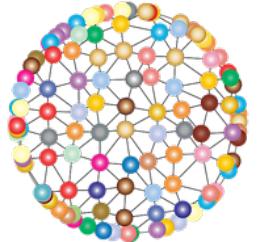
```
'-- src
  |-- com.inden.module1
  |   |-- com
  |   |   '-- inden
  |   |       '-- module1
  |   |           '-- Application.java
  |   '-- module-info.java
  '-- com.inden.module2
      |-- com
      |   '-- inden
      |       '-- module2
      |           '-- OtherClass.java
      '-- module-info.java
```

- In der Praxis oftmals ungeeignet!!!!





**Was ist daran
problematisch?**



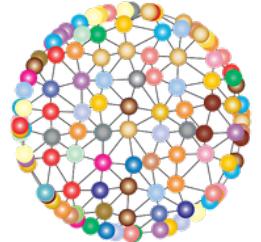


Zwar erlaubt dieses Verzeichnисlayout das Kompilieren in einem Rutsch mit dem neuen Compiler-Feature Multi Module Build, jedoch erschwert es

1. die Trennung der Module in verschiedene Projekte (für Build-Tools und IDEs),
2. das Erstellen von JARs als eigenständige Deployables sowie
3. die saubere Separation der Module untereinander.

Konsequenzen:

- ⇒ Das Ganze stellt die Ziele der Modularisierung ein wenig infrage
 - ⇒ Es sollte ja gerade kleine, in sich abgeschlossene Komponenten bzw. Module erstellen möchte
 - ⇒ die durch die obige Anordnung aber eher wieder zu einem Monolith.
-



Wie geht es besser?



Variante 1: Applikation mit mehreren Modulen => mehrere src-Verzeichnisse mit Modulnamen

```
playlistservice-java9-modules-example
|--- playlistservice
|   '--- src
|     '-- main
|       '-- java
|         '-- playlistservice
|           |-- com
|             '-- javaprofi
|               '-- spi
|                 '-- PlayListService.java
|             '-- module-info.java
|--- playlistserviceconsumer
|   '--- src
|     '-- main
|       '-- java
|         '-- playlistserviceconsumer
|           |-- com
|             '-- serviceconsumer
|               '-- ServiceConsumerExample.java
|             '-- module-info.java
|--- playlistserviceprovider
|   '--- src
```

Einführung: Besseres Verzeichnislayout



Variante 2: Applikation mit mehreren Modulen => mehrere *normale* src-Verzeichnisse

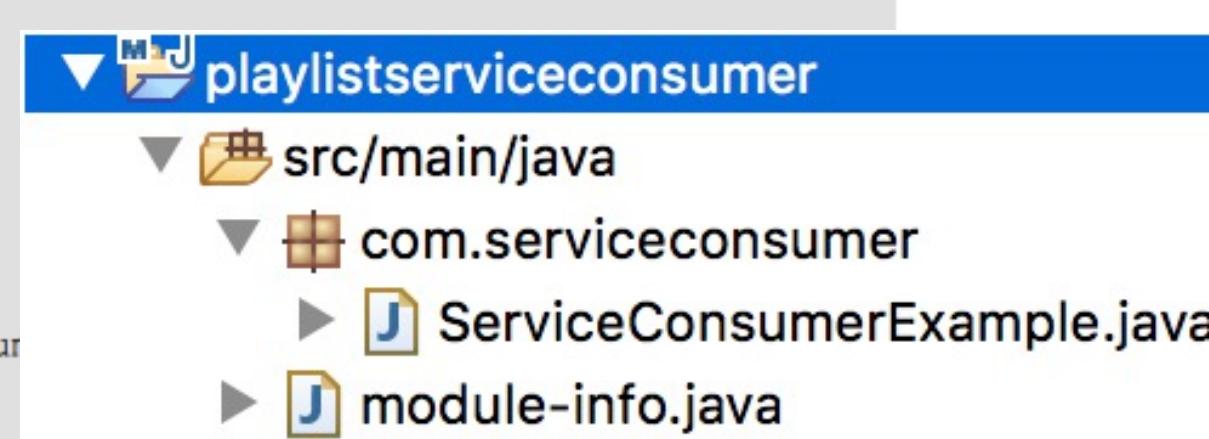
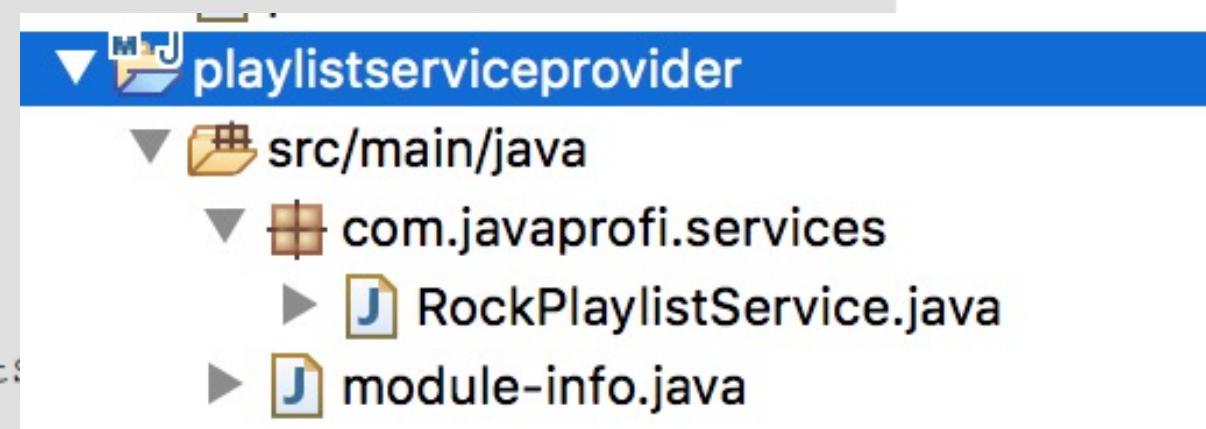
```
playlistservice-java9-modules-example
|--- playlistservice
|   '-- src
|     '-- main
|       '-- java
|         '-- playlistservice
|           |-- com
|             '-- javaprofi
|               '-- spi
|                 '-- PlayListService.java
|               '-- module-info.java
|--- playlistserviceconsumer
|   '-- src
|     '-- main
|       '-- java
|         '-- playlistserviceconsumer
|           |-- com
|             '-- serviceconsumer
|               '-- ServiceConsumerExample.java
|             '-- module-info.java
|--- playlistserviceprovider
|   '-- src
```

Einführung: Besseres Verzeichnislayout



Variante 2: Applikation mit mehreren Modulen => mehrere *normale* src-Verzeichnisse

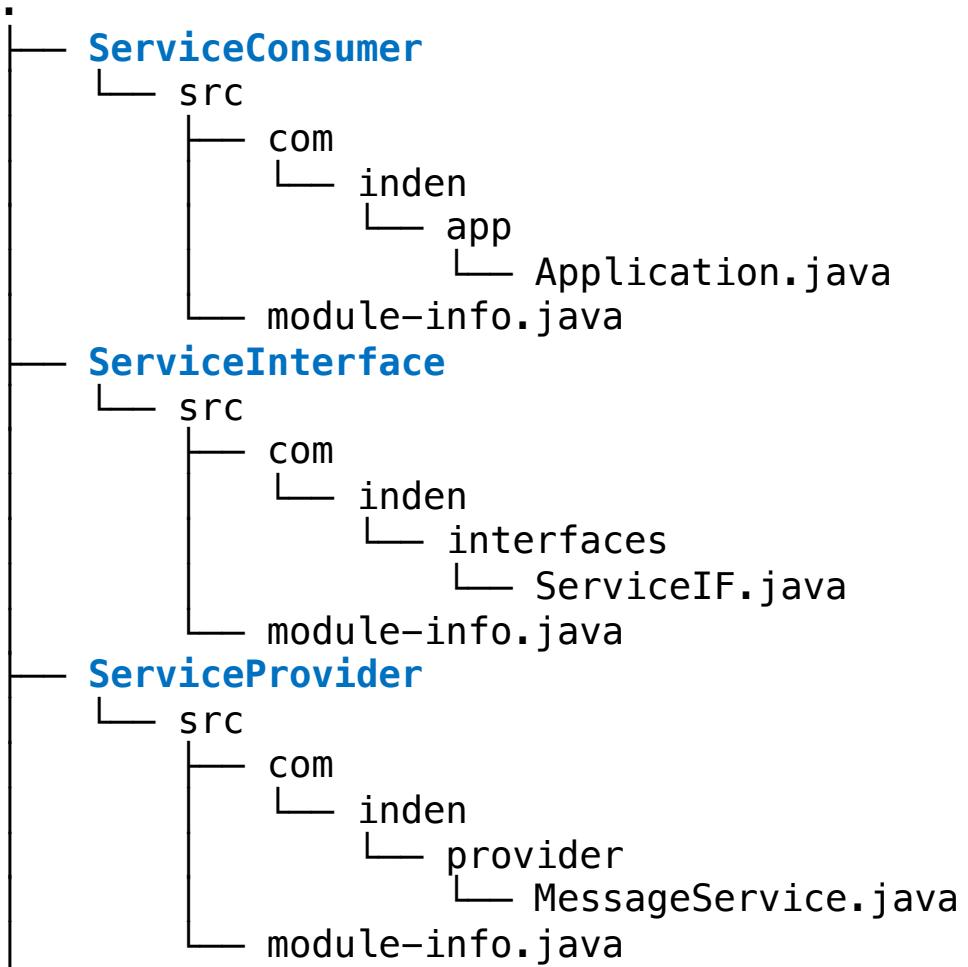
```
playlistservice-jav9-modules-example
|-- playlistservice
|   '-- src
|       '-- main
|           '-- java
|               '-- playlistservice
|                   |-- com
|                       '-- javaprofi
|                           '-- spi
|                               '-- PlayListS...
|                               '-- module-info.java
|
|-- playlistserviceconsumer
|   '-- src
|       '-- main
|           '-- java
|               '-- playlistserviceconsumer
|                   |-- com
|                       '-- serviceconsumer
|                           '-- ServiceConsu...
|                           '-- module-info.java
|
`-- playlistserviceprovider
    '-- src
```



Einführung: Besseres Verzeichnislayout



Beispiel App mit 3 Modulen, hier Eclipse-Variante src und «ohne» Modulname



IDE & Tool Support



- Aktuelle IDEs & Tools grundsätzlich gut
- Eclipse: Module entsprechen Projekten
- IntelliJ: Spezielles Modulkonstrukt unterhalb von Projekten
- Noch kein Standardlayout, diverse Variationen möglich
- Nutze gerne das normale Layout ohne den Modulnamen im «src»
- Für Module ansonsten Konfiguration der Pfade notwendig ☹
- Maven durch Multi-Module-Build etwas komfortabler als Gradle
- Gradle erfordert die manuelle Konfiguration des Module-Path



```
tasks.withType(JavaCompile) {  
    options.compilerArgs += ["--module-path", classpath.asPath]  
}
```

The Maven logo, featuring the word "Maven" in a bold, black, sans-serif font with a colorful feather icon above the letter 'e'.

The Gradle logo, featuring a stylized blue elephant icon to the left of the word "Gradle" in a bold, blue, sans-serif font.



PART 2

Sichtbarkeiten und transitive Abhängigkeiten





Sichtbarkeiten



Sichtbarkeiten in JDK 8

- **private** – Lediglich in der eigenen Klasse sichtbar
- **Default** – sichtbar im Package
- **protected** – Wie default, aber auch für abgeleitete Klassen sichtbar (auch in anderen Packages)
- **public** – Sichtbar für alle Klassen

=> **PUBLIC** = sichtbar für jeder im CLASSPATH!!

=> Eigentlich KEINE Zugriffskontrolle

Sichtbarkeiten

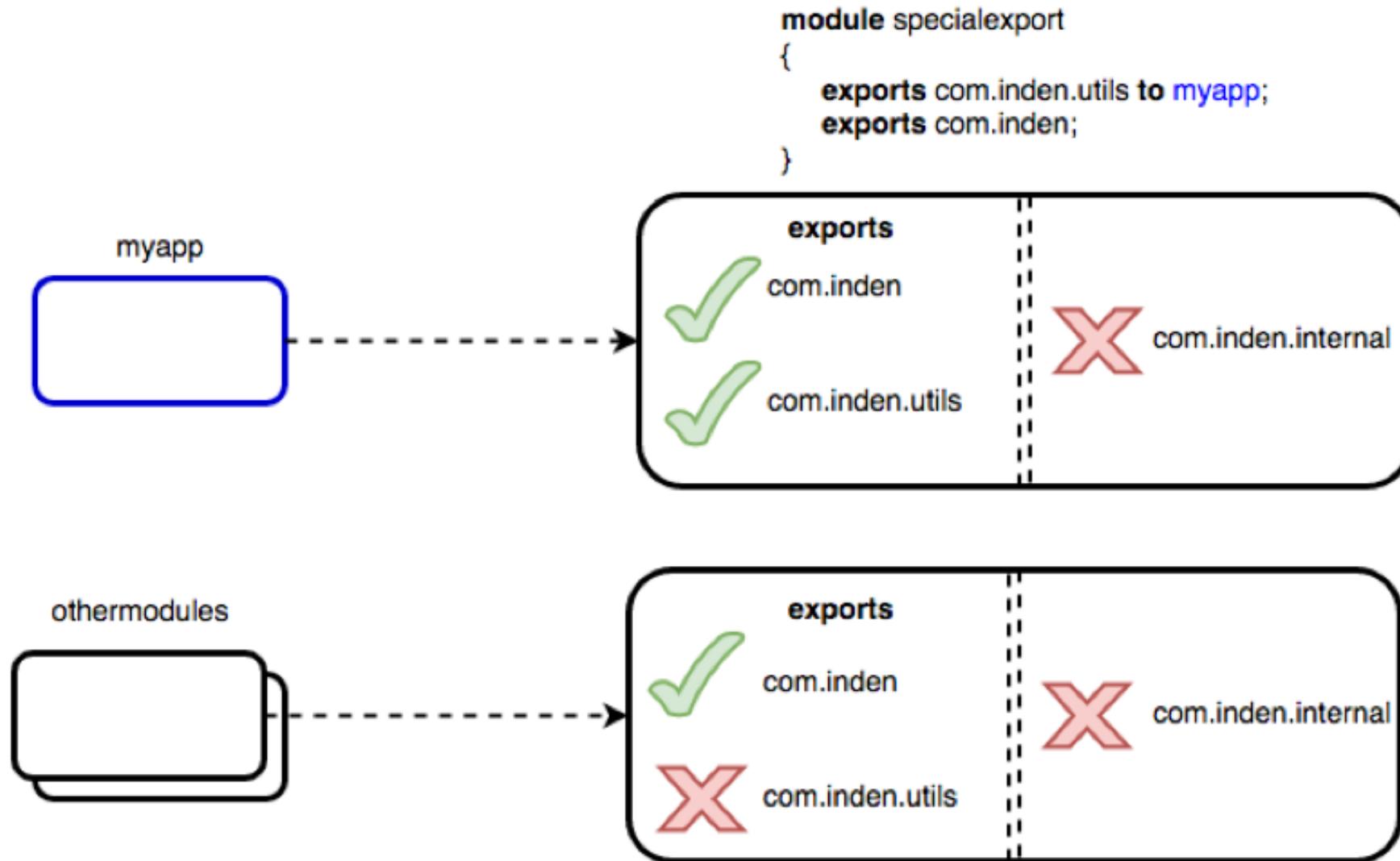


Sichtbarkeiten in JDK 9

Mit der Einführung der Modularisierung kann die Sichtbarkeit von Typen genauer spezifiziert werden. Dies ist in der Regel nur für Typen relevant, die `public` definiert sind.

- **Exported Globally (exports + requires)** – öffentlich für alle Module, die darauf verweisen (`requires`)
- **Qualified export (exports to + requires)** – just visible for specified modules and those who are reading
- **Modul intern (kein export)** – nur im Modul selbst sichtbar

Sichtbarkeitssteuerung





Qualified Export (export ... to ...)

- in den Moduldeskriptoren nicht freigegebene Zugriffe werden verhindert.
- ein eingeschränkter Export für eine definierte Menge an Modulen möglich

```
module java.base
{
    exports sun.reflect to java.corba,
             java.logging,
             java.sql,
             java.sql.rowset,
             jdk.scripting.nashorn;
}
```

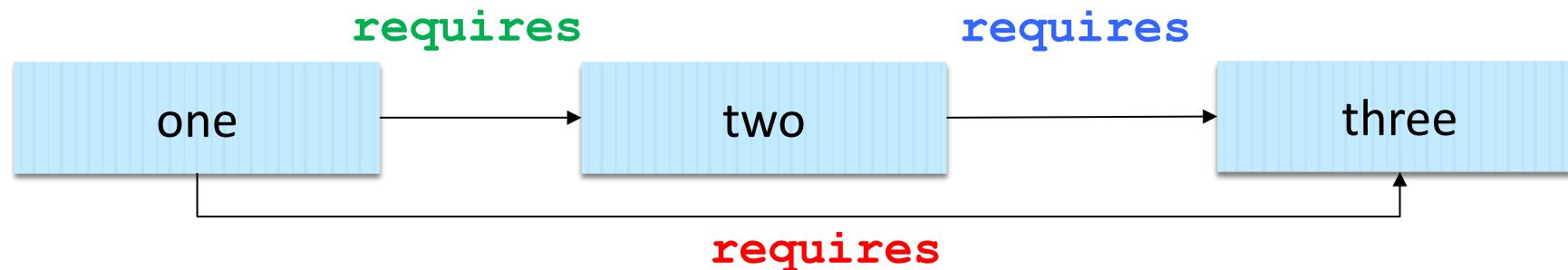


Transitive Abhangigkeiten (Implied Readability)

Transitive Abhängigkeiten



- Die in Moduldeskriptoren beschriebenen Abhängigkeiten werden nicht automatisch an nutzende Module propagiert.
- Immer dann, wenn mehrere Module kombiniert werden, kann dies umständlich werden



```
public class A {  
    public void bar() {  
        B b = new B();  
        b.foo(); // ok  
  
        // requires three  
        C c = b.foo();  
    }  
}
```

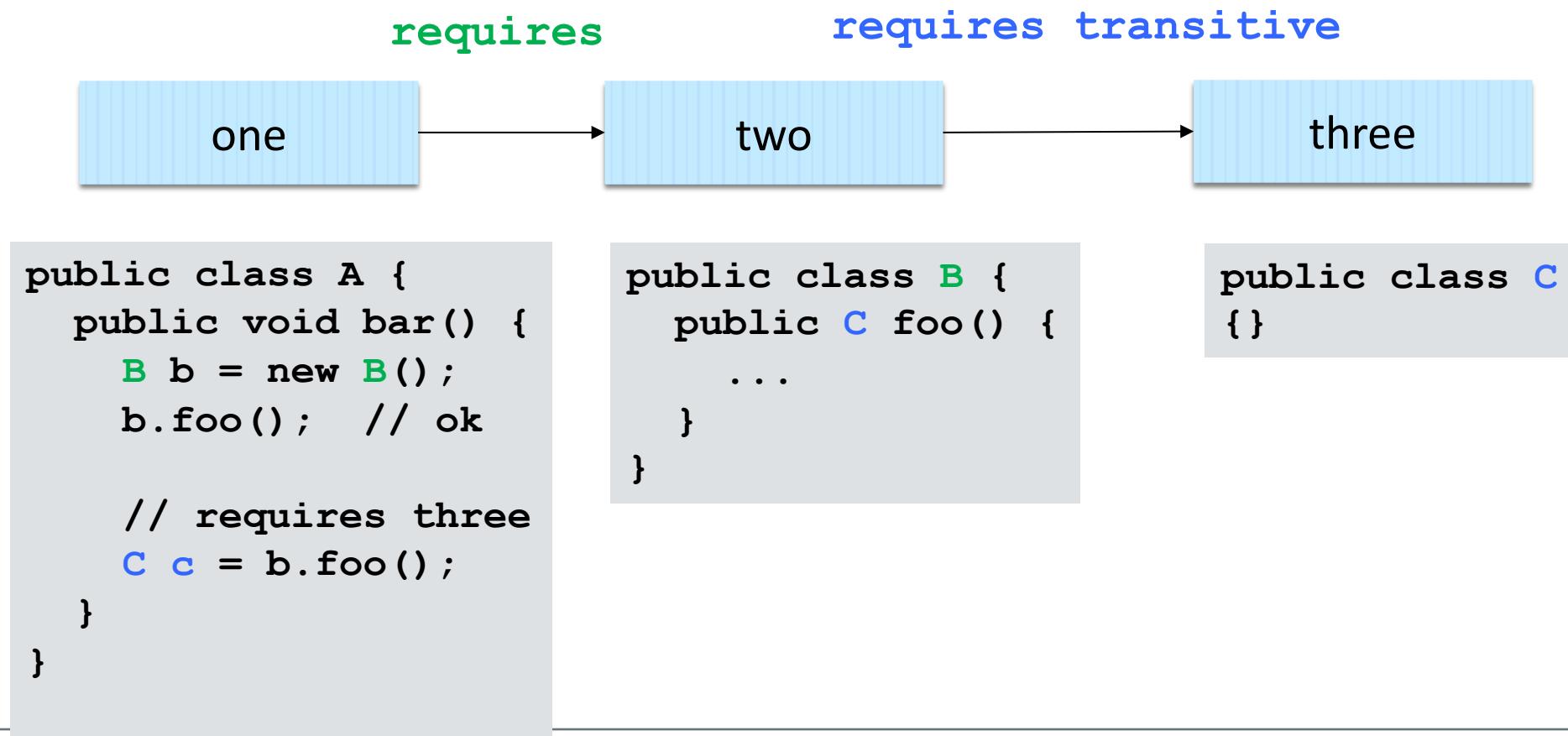
```
public class B {  
    public C foo() {  
        ...  
    }  
}
```

```
public class C  
{}
```

Transitive Abhängigkeiten



transitive Abhängigkeit kann explizit als solche deklariert werden



Transitive Abhängigkeiten



Readability in the Java SE module graph

```
module java.sql {  
    requires java.logging;  
    exports java.sql;  
}
```

```
package java.sql;  
import java.util.logging.Logger;  
public interface Driver {  
    Logger getParentLogger();  
}
```

```
module java.logging {  
    exports java.util.logging;  
}
```

```
package java.util.logging;  
public class Logger {  
    ...  
}
```



Readability in the Java SE module graph

```
module myApp {  
    requires java.sql;  
    requires java.logging; ☺  
}  
  
module java.sql {  
    requires java.logging;  
    exports java.sql;  
}  
  
module java.logging {  
    exports java.util.logging;  
}
```



Readability in the Java SE module graph

```
module myApp {  
    requires java.sql;  
    requires java.logging; ☺  
}
```

```
module java.sql {  
    transitive requires public java.logging;  
    exports java.sql;  
}
```

```
module java.logging {  
    exports java.util.logging;  
}
```

Transitive Abhängigkeiten



Implied Readability (Aggregator-Module)

- Oftmals ist es praktisch, verschiedene **Module** zu größeren Einheiten zu bündeln.
Das ist mithilfe von **requires transitive** möglich.

```
java.se@9-ea
  requires mandated java.base
  requires transitive java.compiler
  requires transitive java.datatransfer
  requires transitive java.desktop
  requires transitive java.instrument
  requires transitive java.logging
  requires transitive java.management
  requires transitive java.management.rmi
  requires transitive java.naming
  requires transitive java.prefs
  requires transitive java.rmi
  requires transitive java.scripting
  requires transitive java.security.jgss
  requires transitive java.security.sasl
  requires transitive java.sql
  requires transitive java.sql.rowset
  requires transitive java.xml
  requires transitive java.xml.crypto
```



PART 3

Abhängigkeiten mit Services lösen

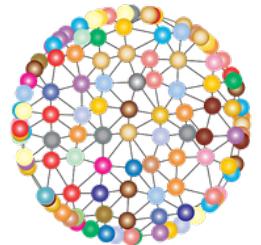


Modul-Abhängigkeiten



Module erlauben das **Untergliedern** einer Applikation in mehrere Bestandteile:

- **Aber bisher: Ein Modul verweist direkt auf ein anderes bzw. genauer eine Klasse nutzt eine Klasse eines anderen Moduls direkt.**
- **Unproblematisch für die Nutzung von Klassen des JDKs**
- **Mitunter fragwürdig bei eigenen Applikationen => stärkere Implementierungsabhängigkeit**

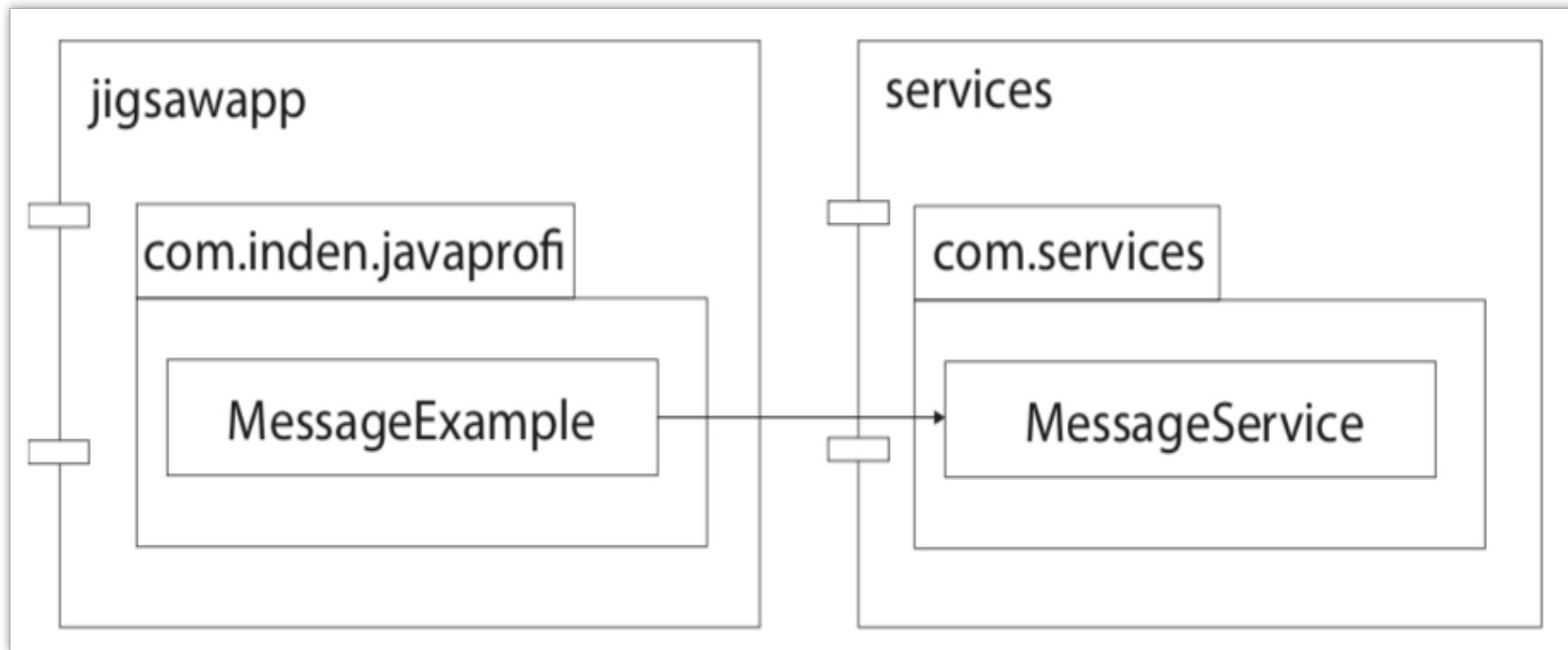


**Was ist daran
problematisch?**

Enge Kopplung



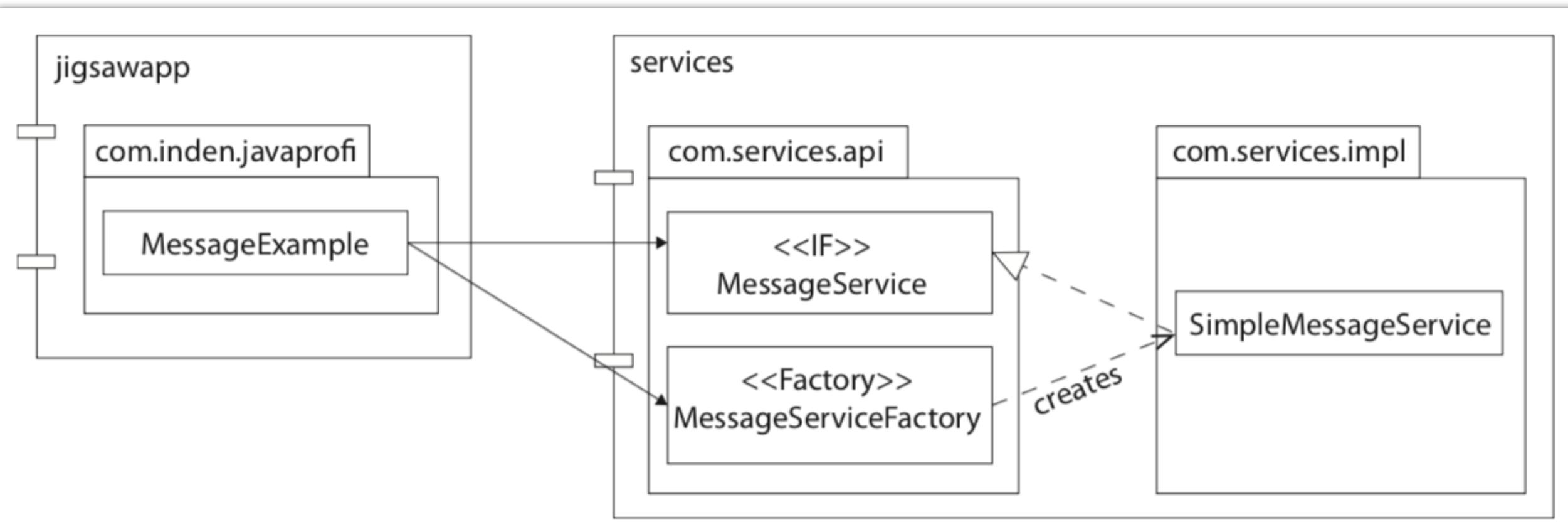
Existiert eine direkte Verbindung zwischen den Modulen, so sind diese eng miteinander gekoppelt und bilden **logisch eigentlich ein Modul**.



Entkopplung erzielen (mit Factory)



Soll eine stärkere Entkopplung erfolgen, so verweist eine nutzende Applikation idealerweise lediglich auf Interfaces und erhält die Realisierungen über Factory-Methoden oder Services.

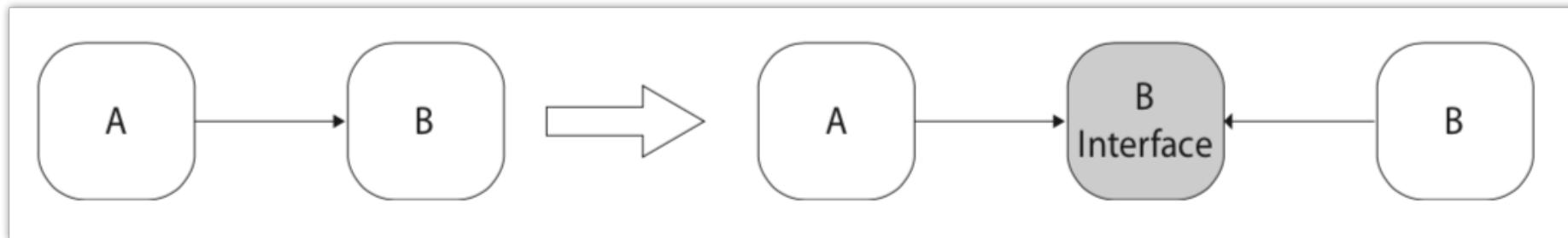


Entkopplung erzielen (mit Factory)



Was erreichen wir durch die Einführung der Indirektion und der Factory?

1. Wir sind funktional immer noch auf demselben Stand wie zuvor, aber keine (starke) Implementierungsabhängigkeit auf wichtige konkrete Klassen mehr – es verbleibt lediglich die Abhängigkeit auf die Factory-Klasse und das Interface.
2. Eine weiterführende Variante = zusätzliches Modul mit Interface-Definitionen, auf das dann beide Module verweisen ([Dependency Inversion Principle](#))

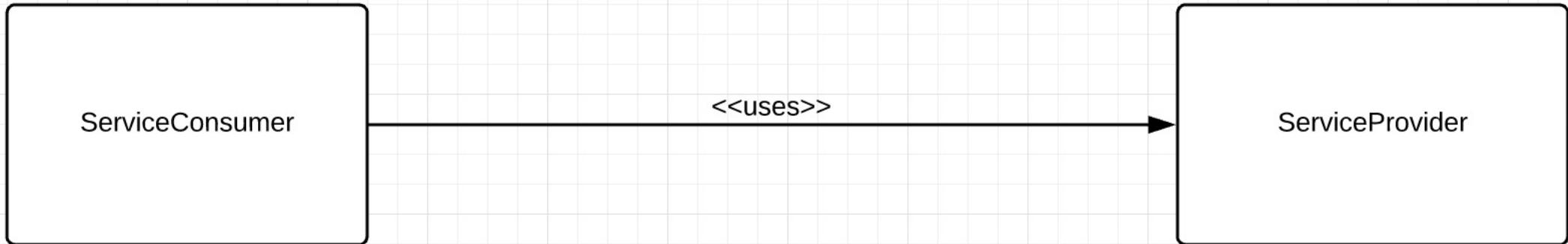


3. Eine Variante, wie man die Kopplung noch weiter reduziert, sind [Services](#).

Services – Bestandteile



Direkte Kopplung



Losere Kopplung über Interface





Services erfordern 3 Schritte:

1. Definition eines **Service Interface**
 2. Definition eines **Service Provider**
 3. Definition eines **Service Consumer**
- 3b Abstraktion der Zugriffe und Bereitstellung mithilfe der Klasse **ServiceLoader**



- Die Klasse **java.util.ServiceLoader**

- ermöglicht losere Kopplung
- erlaubt es, verschiedene Realisierungen eines Interface oder einer abstrakten Klasse als Service bzw. Erweiterung erst zur Laufzeit einzubinden.
- Implementierungen werden zur Laufzeit geladen und als Iterator<T> bereitgestellt:

```
final Iterator<DesiredServiceInterface> iterator =  
    ServiceLoader.load(DesiredServiceInterface.class).iterator();
```

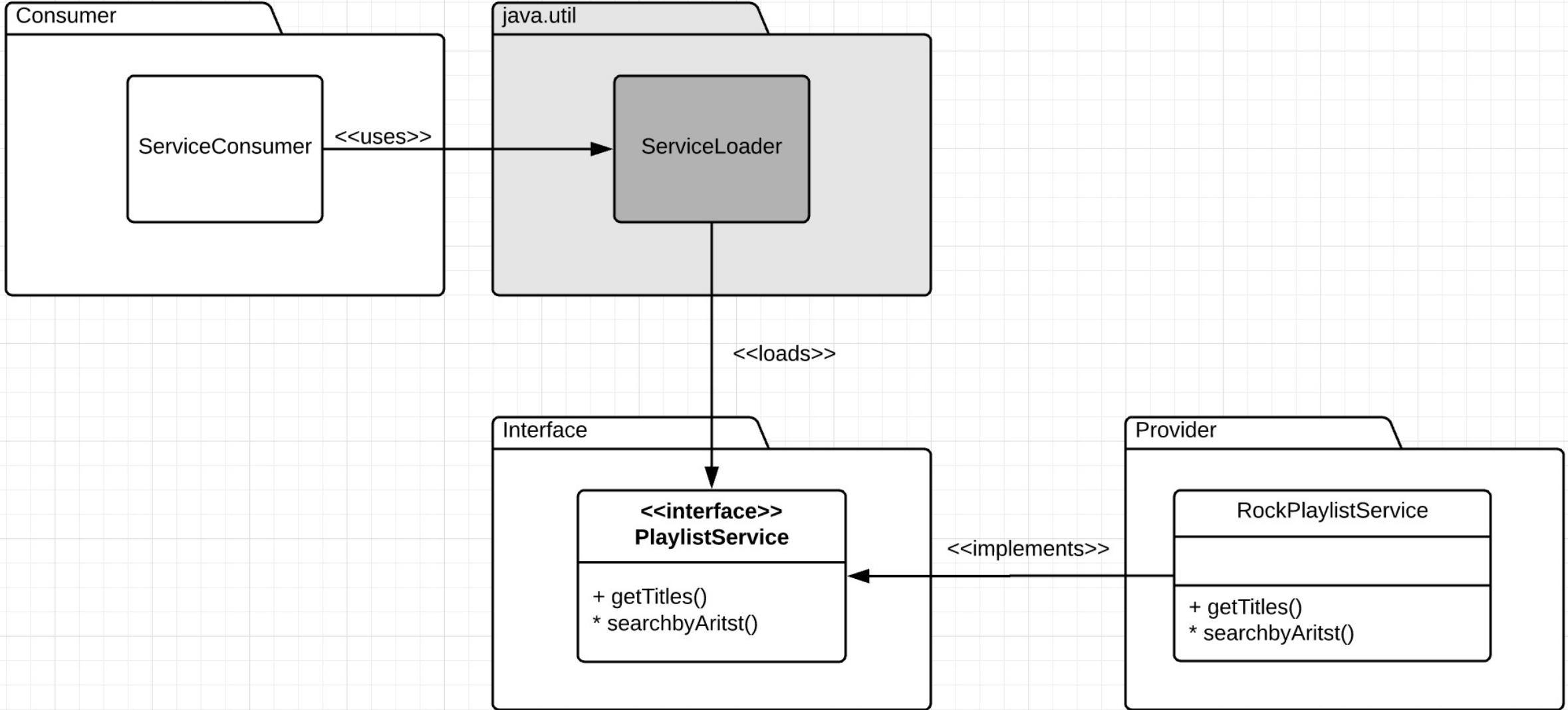
- Dazu sucht der **ServiceLoader** den CLASSPATH ab. Für Module gibt es spezielle Mechanismen zur Steuerung in den Moduldeskriptoren.



DEMO

ServiceLoaderExample

Services – Zu realisierende Applikation



Services – Schritt 1: Service Interface



Services erfordern 3 Schritte:

1. Definition eines Service Interface

```
public interface PlaylistService
{
    public List<String> getTitles();
    public List<String> searchByArtist(final String artist);
}
```

2. Definition eines Service Provider
 3. Definition eines ServiceConsumer
-

Services – Schritt 2: Definition eines Service Provider



```
public class RockPlaylistService implements PlaylistService
{
    private final Map<String, List<String>> songMap = Map.of("Bryan Adams", List.of("Summer of '69"),
        "Bon Jovi", List.of("Livin' On A Prayer"), "Metallica", List.of("Nothing Else Matters"),
        "Nickelback", List.of("How You Remind Me"), "Toto", List.of("Africa", "Hold The Line"));

    @Override
    public List<String> getTitles()
    {
        final Stream<List<String>> titlesStream = songMap.values().stream();
        return titlesStream.reduce(new ArrayList<>(), (a, b) -> {a.addAll(b); return a;});
    }

    @Override
    public List<String> searchByArtist(final String artist)
    {
        return songMap.getOrDefault(artist, List.of("No title found for " + artist));
    }
}
```

- muss **public** sein und einen **No-Arg-Konstruktor** besitzen, damit der **ServiceLoader** die Klasse laden und per **Reflection** instanziieren kann.

Services – Schritt 3: ServiceConsumer & Zugriff über den ServiceLoader



```
public static void main(final String[] args) throws Exception
{
    final Optional<PlaylistService> optService = lookup(PlaylistService.class);

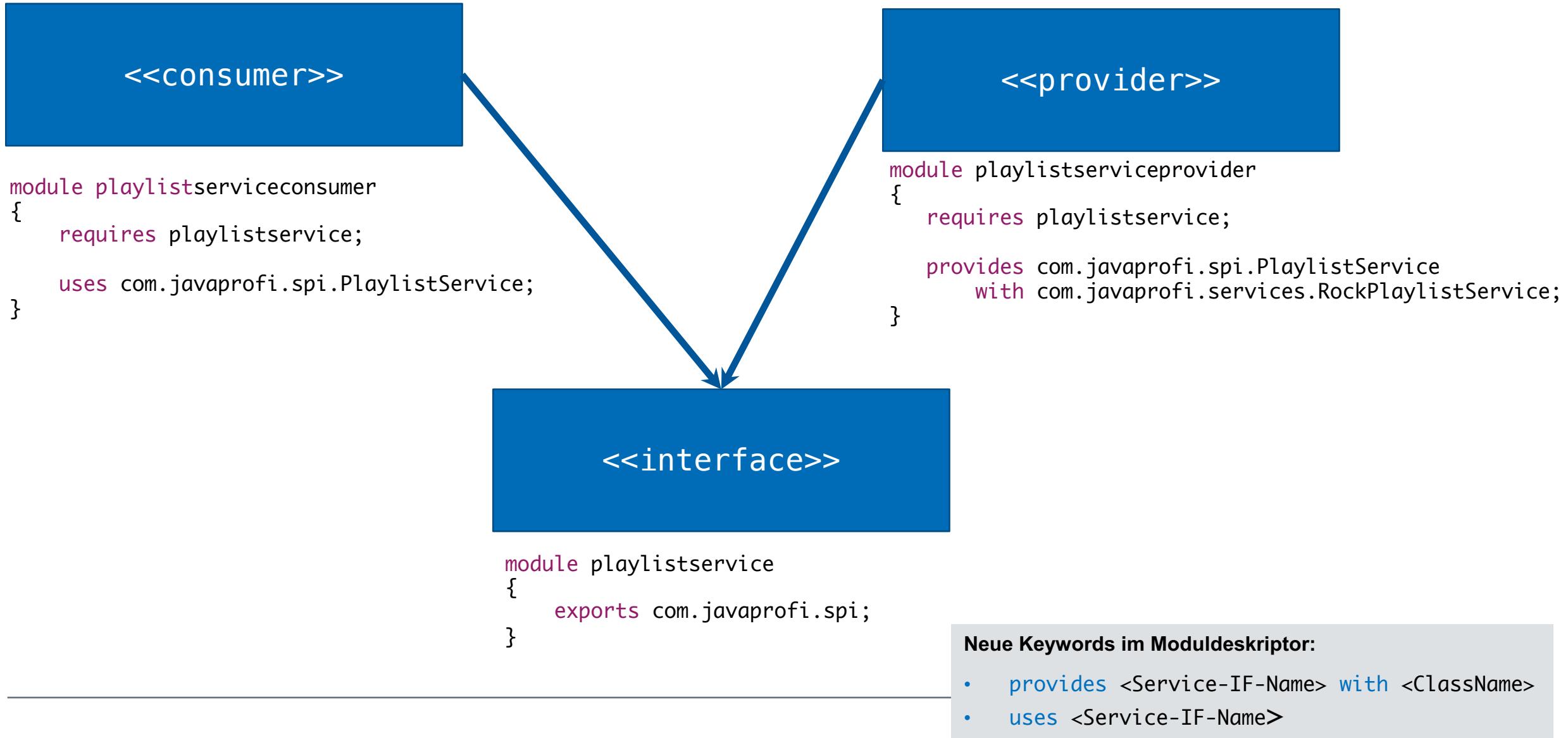
    optService.ifPresentOrElse(service -> useService(service),
                               () -> System.err.println("No service provider found!"));
}

private static void useService(final PlaylistService service)
{
    System.out.println(service.getClass());

    final List<String> allTitles = service.getTitles();
    System.out.println("All titles: " + allTitles);
}

private static <T> Optional<T> lookup(final Class<T> clazz)
{
    final Iterator<T> iterator = ServiceLoader.load(clazz).iterator();
    return iterator.hasNext() ? Optional.of(iterator.next()) : Optional.empty();
}
```

Services – Mit Modularisierung / Zielarchitektur



Services – Schritt 1: Definition eines Service Interface



a) Definition eines Service Interface (analog zu vorher)

```
package com.javaprofi.spi;

import java.util.List;

public interface PlaylistService
{
    public List<String> getTitles();
    public List<String> searchByArtist(final String artist);
}
```

b) Definition des Moduldeskriptors

```
module playlistservice
{
    exports com.javaprofi.spi;
}
```

Services – Schritt 2: Definition eines Service Provider



a) Implementierung des Service Provider (analog zu vorher)

```
package com.javaprofi.services;  
...  
  
import com.javaprofi.spi.PlaylistService;  
  
public class RockPlaylistService implements PlaylistService  
{ ... }
```

b) Definition des Moduldeskriptors

```
module playlistserviceprovider  
{  
    requires playlistservice;  
  
    provides com.javaprofi.spi.PlaylistService  
        with com.javaprofi.services.RockPlaylistService;  
}
```

Services – Schritt 3: Nutzung eines ServiceLoaders



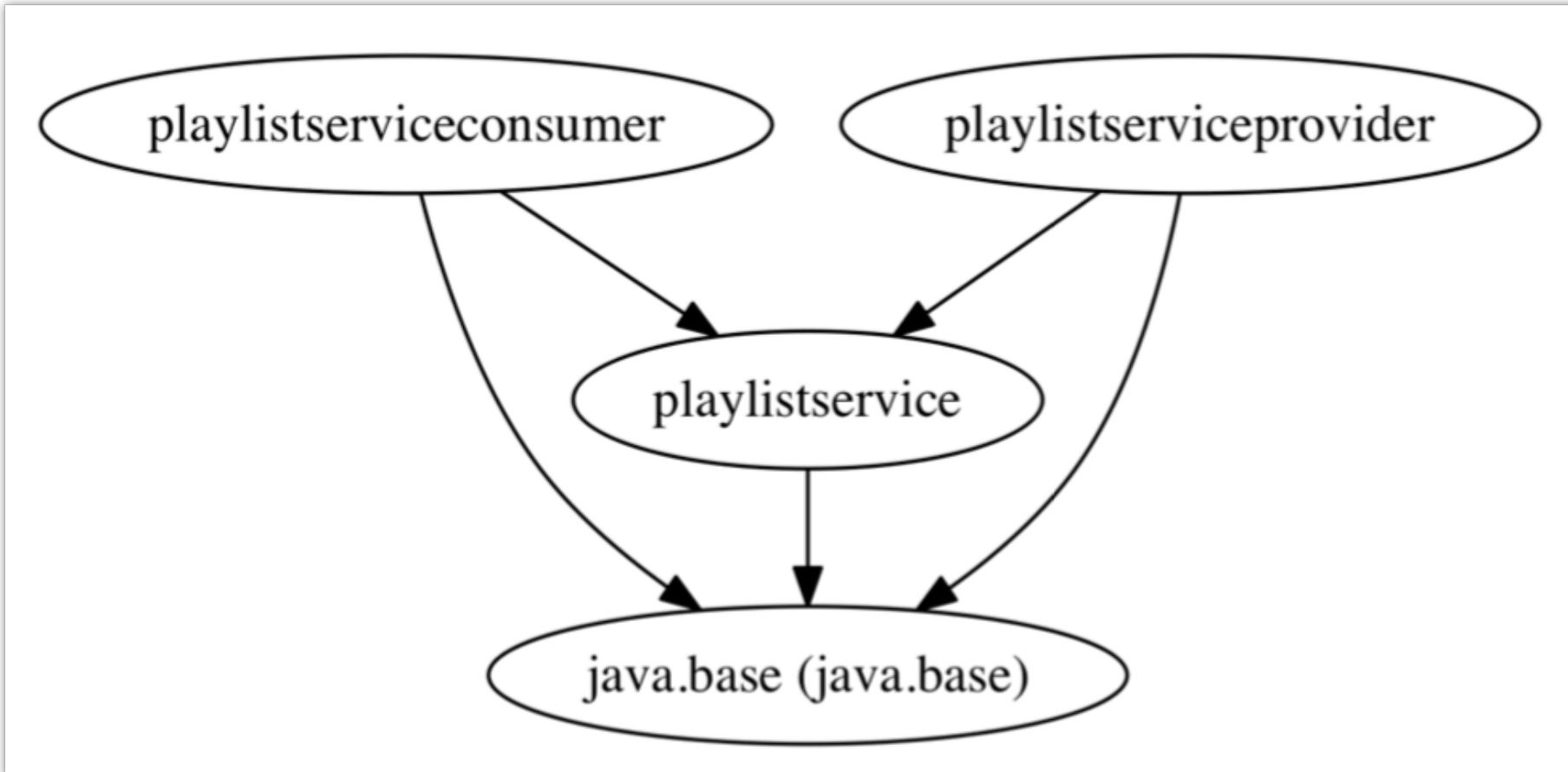
a) Implementierung des Service Consumers (analog zu vorher)

```
package com.serviceconsumer;  
...  
  
import com.javaprof.spi.PlaylistService;  
  
public class ServiceConsumer  
{ ... }
```

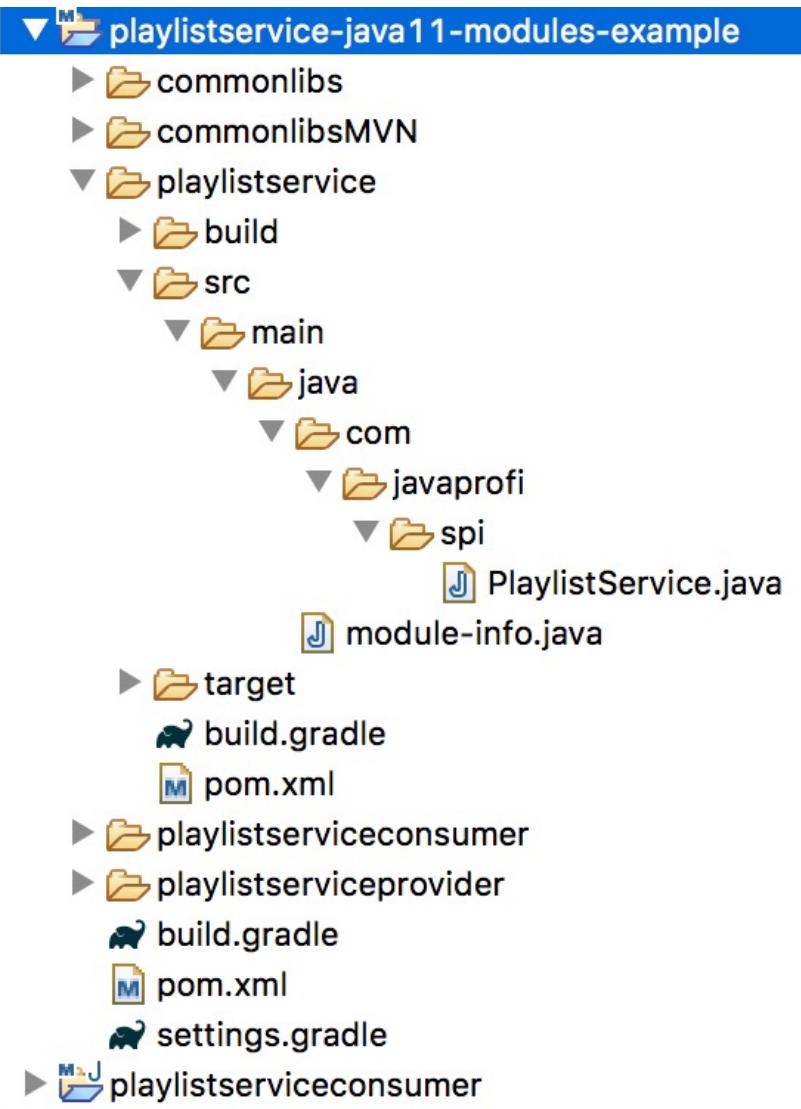
b) Definition des Moduldeskriptors

```
module playlistserviceconsumer  
{  
    requires playlistservice;  
  
    uses com.javaprof.spi.PlaylistService;  
}
```

Services – Kontrolle der Abhangigkeiten



Demo PlaylistService



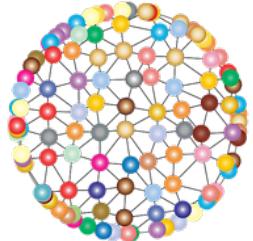


PART 4: Externe Module einbinden / Migrationen





Wie binde ich andere JARs ein?





3rd Party JARs





- **Viele Bibliotheken sind noch nicht für Jigsaw ausgelegt! Was nun?**
 - Kompatibilitätsmodus alles aus CLASSPATH
 - Migration mit **Automatic Modules**
- **Automatic Modules entstehen, wenn herkömmliches JAR im Module-Path aufgeführt wird. Beispiel guava-22.0.jar**

```
module mymodule
{
    requires guava;
}
```



Exkurs Modularten



- **Named Platform Modules** – Bekanntermaßen wurde im Rahmen von Project Jigsaw auch das JDK in Module unterteilt. Diese speziellen Module des JDks haben **keinen** Zugriff auf den Module-Path.
- **Named Application Modules** – sind Module, die Anwendungen oder Bibliotheken bündeln. JARs, die einen Moduldeskriptor in enthalten. Zugriff auf Module-Path, nicht jedoch auf Klassen aus dem CLASSPATH.
- **Open Modules** – Wie die beiden ersten Module, geben allerdings alle Packages für Reflection nach außen frei.



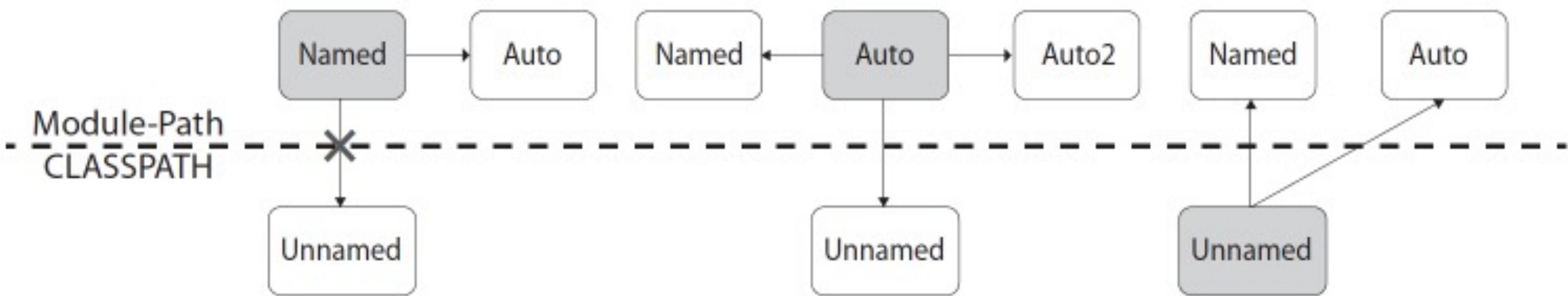
- **Automatic Modules** – Gewöhnliche JAR-Dateien, also ohne module-info.class. Es wird der JAR-Dateiname ohne Versionsnummer und Endung als Modulname genutzt. *Automatic Modules exportieren alle ihre Packages und können auf sämtliche Module aus dem Module-Path sowie JARs aus dem CLASSPATH zugreifen.*
 - **Unnamed Modules** – Ergänzend zum Module-Path kann man sowohl beim Kompilieren als auch beim Programmstart einen CLASSPATH angeben. Alle dort vorhandenen Typen werden zu dem sogenannten Unnamed Module zusammengefasst.
-

Modularten III



Modultyp	Origin	Exports	Readability
Platform	JDK	Via exports	-/-
Application	Modular JAR	Via exports	Platform, Application, Automatic
Automatic	JAR without module descriptor	all packages	Platform, Application, Automatic, Unnamed
Unnamed	JAR from CLASSPATH	all packages	Platform, Application, Automaticd, Unnamed

Zugriffsmöglichkeiten verschiedener Modularten





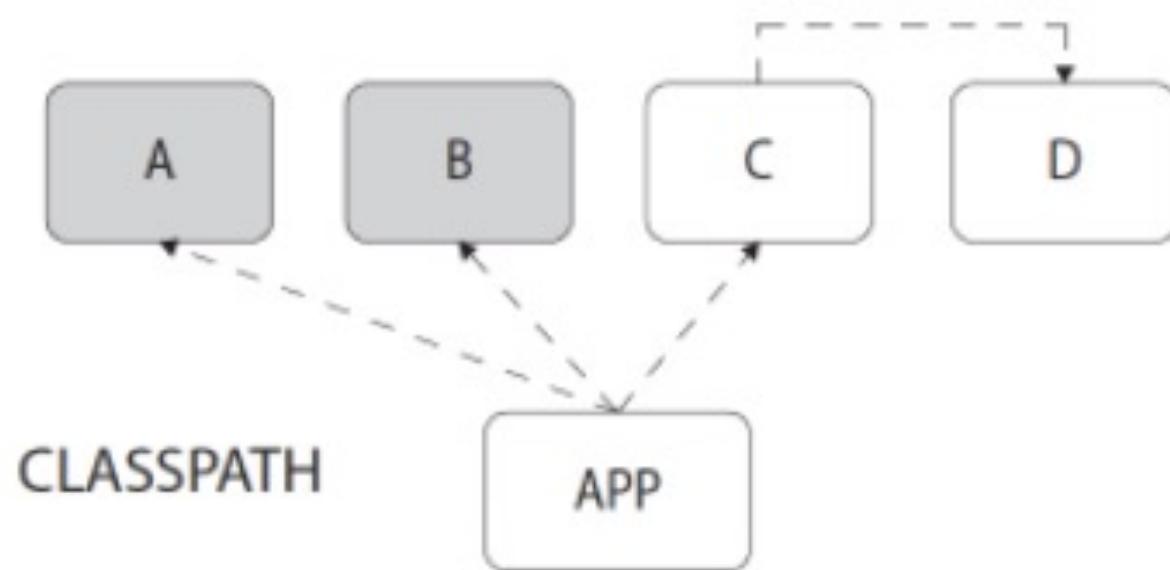
Migrationen



Migration: 3rd Party Bibliotheken einbinden



Betrachten wir eine Migration einer Applikation bestehend aus einigen JARs im CLASSPATH, etwa app.jar , A.jar , B.jar und C.jar:

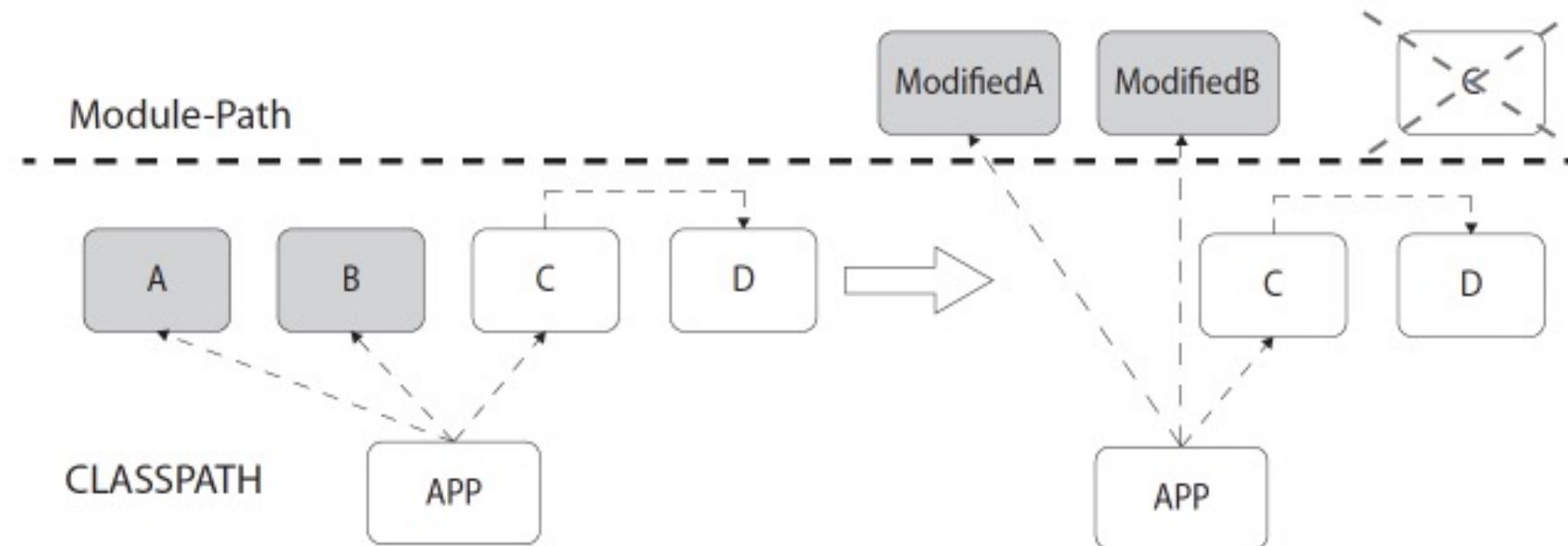




Bei einer sogenannten Bottom-up-Migration werden diese JARs schrittweise in modulare JARs umgewandelt. Dabei kann man folgende zwei Szenarien unterscheiden:

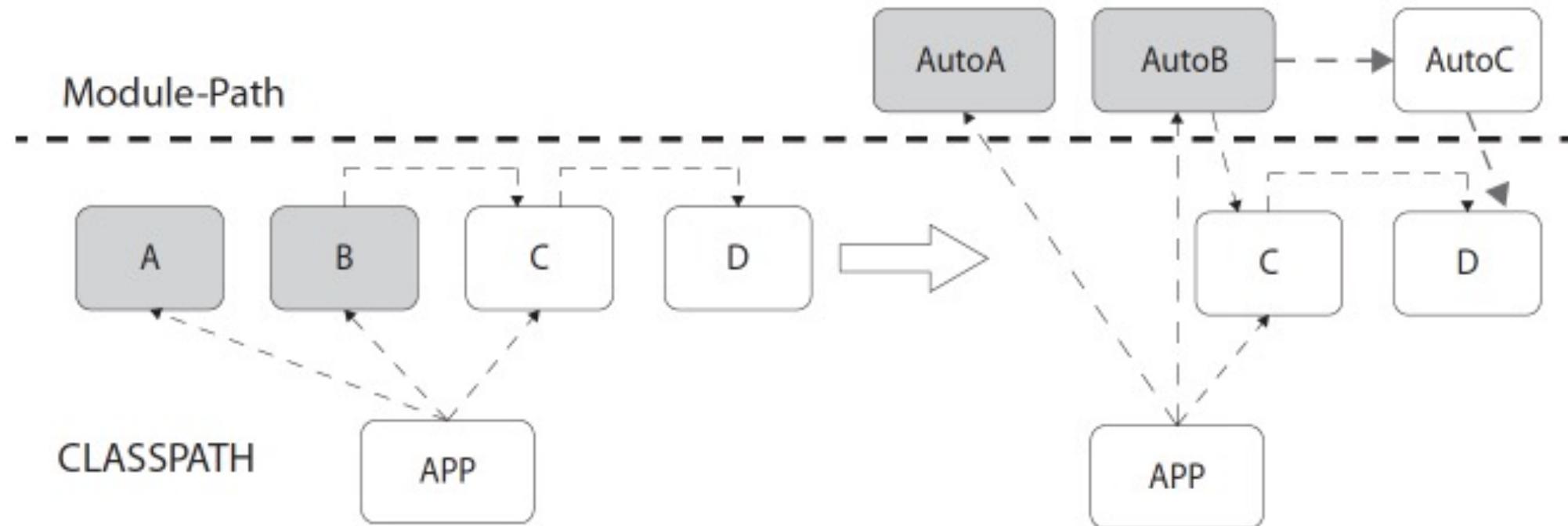
1. **Bottom-up-Migration und Named Modules** – Wenn wir den Sourcecode eines JARs im Zugriff haben, können wir ein JAR in ein modulares JAR überführen, indem wir einen geeigneten Moduldeskriptor hinzufügen und das Modul neu kompilieren und paketieren.
2. **Bottom-up-Migration und Automatic Modules** – Besteht kein Zugriff auf den Sourcecode des JARs, so lässt sich das JAR nicht auf die zuvor beschriebene Weise in ein modulares JAR konvertieren. Als gute Alternative bleibt die Nutzung als Automatic Module, gegebenenfalls in Kombination mit dem Unnamed Module.

Bottom-up-Migration und Named Modules



Wir können ein JAR nur in ein modulares JAR umwandeln, wenn dieses keine Abhängigkeiten auf andere, nicht modulare JARs des CLASSPATH besitzt, weil sich diese nicht im Moduldeskriptor referenzieren lassen.

Bottom-up-Migration und Automatic Modules





Für eine Migration bieten sich folgende Schritte an:

1. **Keine Abhängigkeiten** – Man beginnt mit JARs ohne externe Abhängigkeiten. Sofern deren Sourcecode zur Verfügung steht, lassen sich diese per Bottom-up-Migration in Named Modules überführen – bei Bedarf können diese immer noch als Automatic Modules eingebunden werden.
2. **Mit Abhängigkeiten** – Vorhandene JARs ohne Zugriff auf deren Sourcecode lassen sich problemlos als Automatic Modules nutzen.
3. **Eigene Applikation** – Nachdem die JARs in Named Modules oder Automatic Modules überführt wurden oder aber im Unnamed Module verbleiben, kann man versuchen, die eigene Applikation in ein Named Module zu transformieren.

Bedenke: Nicht jede Applikation lässt sich sinnvoll modularisieren!
Behalte Business Value im Auge!



Questions?





Thank You
