

Workshop Modularisierung mit Project Jigsaw

**Michael Inden
ASMIQ AG**



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre bei Zühlke Engineering AG in Zürich
- Seit Juni 2017 bei **Direct Mail Informatics / ASMIQ** in Zürich
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@asmiq.ch

Kursangebot: <https://asmiq.ch/>

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



Agenda

-
- **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
 - PART 5: Reflection
 - **Fazit**

PART 1: Einführung

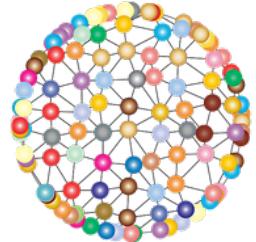


-
- **Warum Modularisierung?**
 - **Was ist ein Modul?**
 - **Modularisierung des JDKs?**
 - **Abhängigkeiten und Sichtbarkeiten steuern**
-

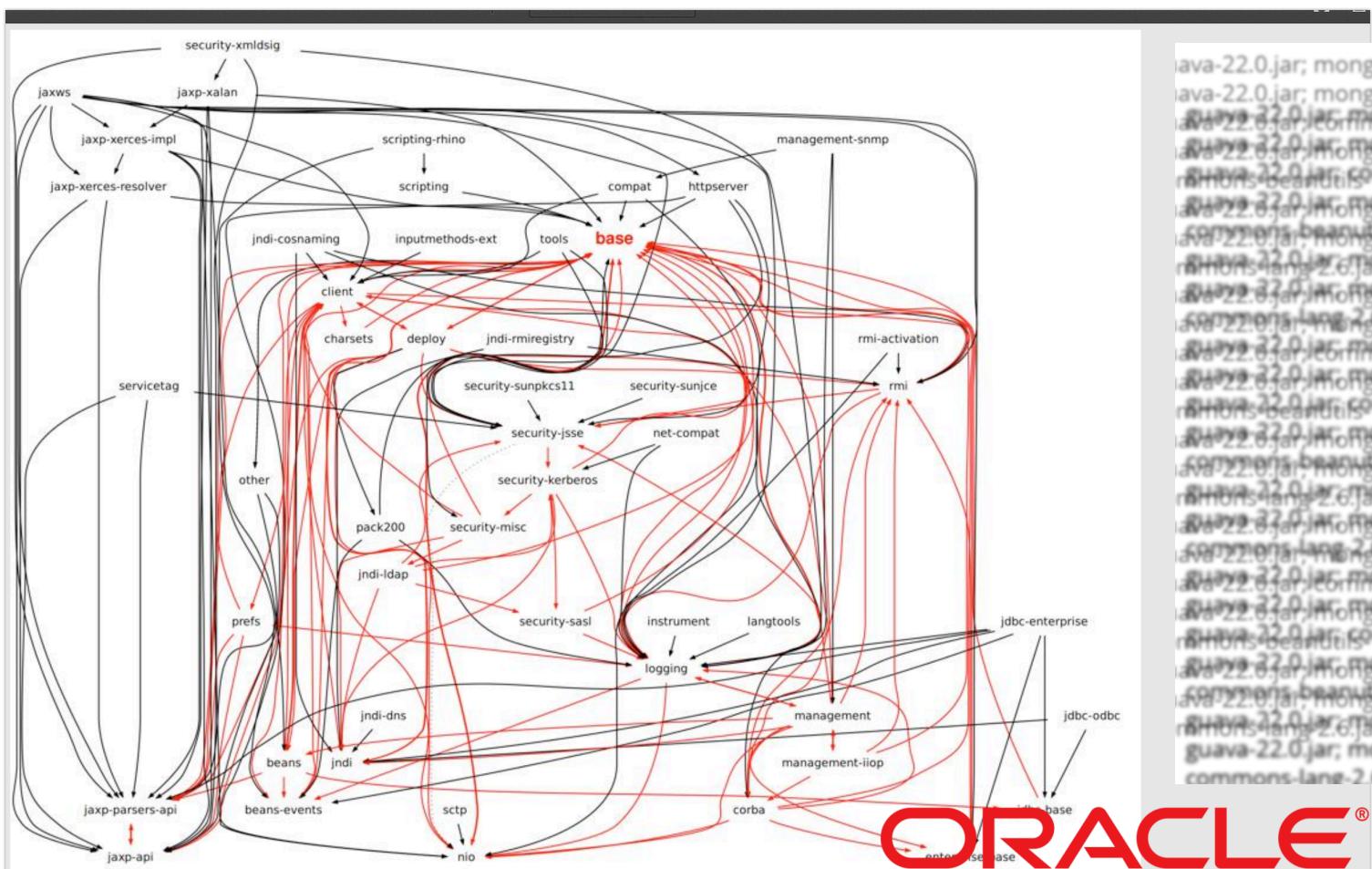
- **Modularisierung von Anwendungen und Bibliotheken ermöglichen**
- **Modularisierung des JDKs an sich**
- **zuverlässige Konfiguration statt fehlerträchtiger Abhängigkeitsverwaltung**



**Aber warum ist das
erstrebenswert?**



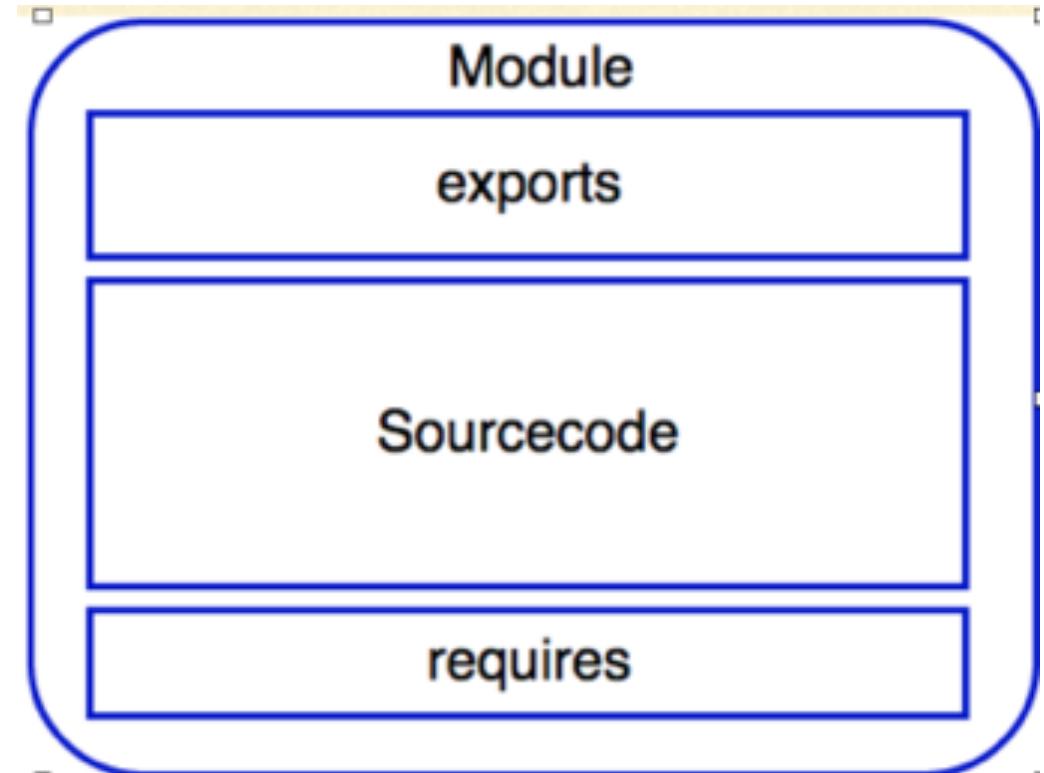
- Situation mit Java 8: **Wirre Abhangigkeiten** und **Chaos im CLASSPATH**



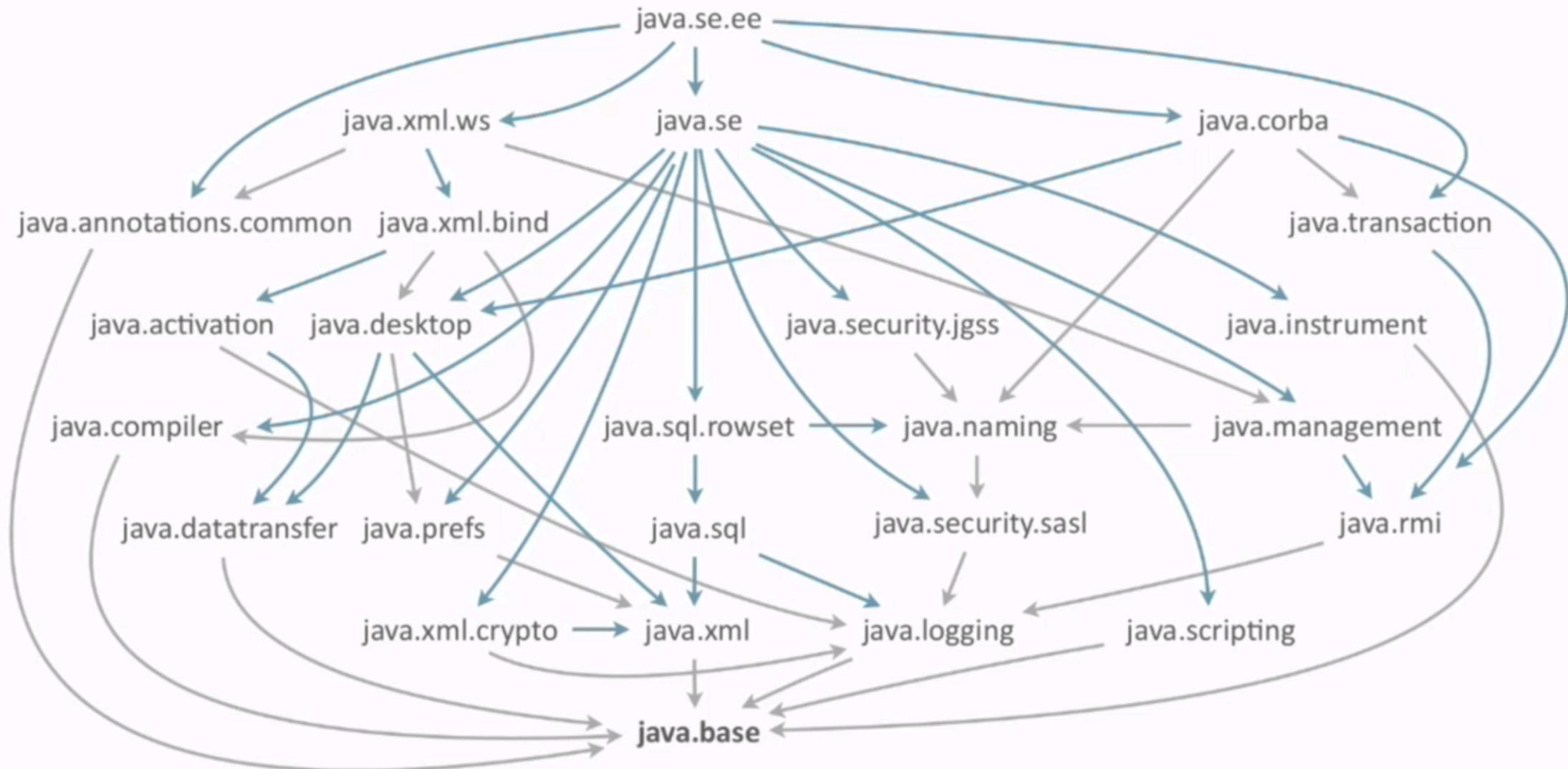
Situation mit Java 8:

- „**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

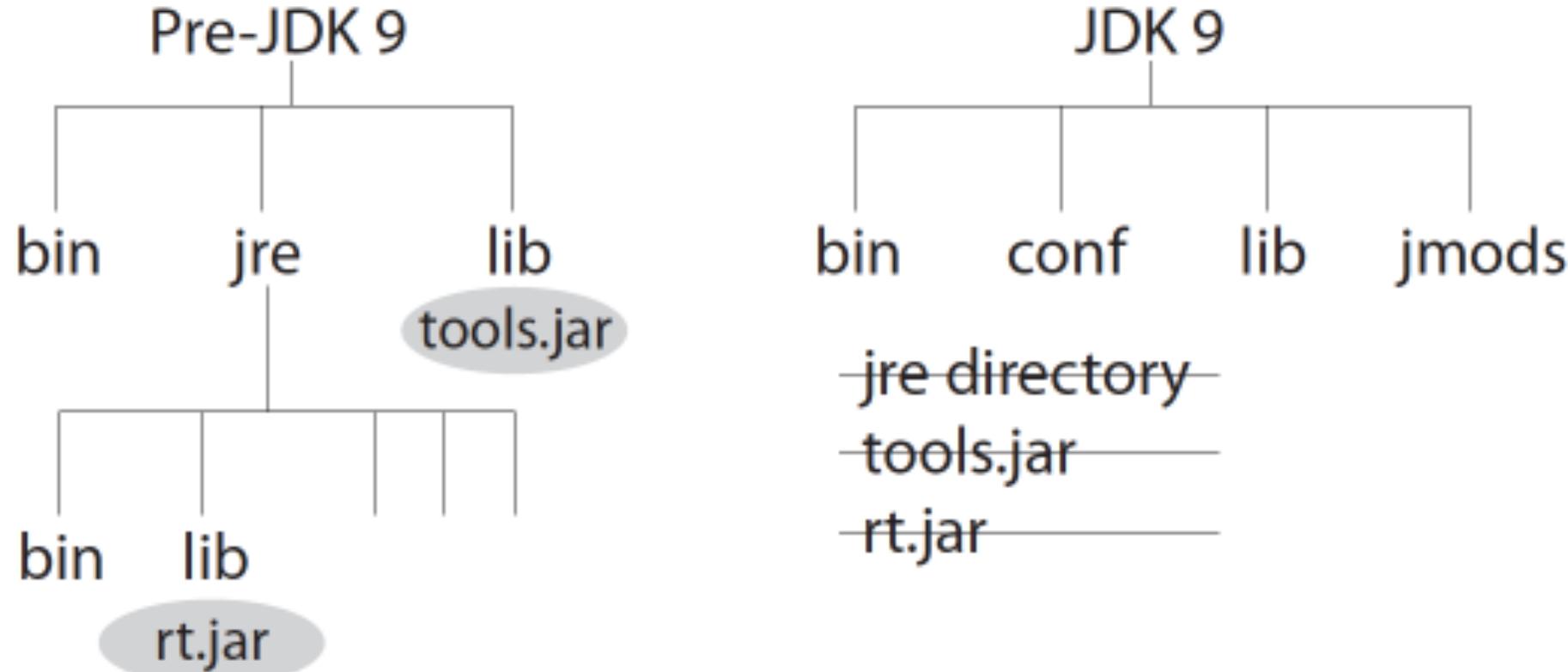
- **Module als neue Softwarebausteine im JDK als Gruppierung von Packages**
- **Ein Modul ...**
 - **besitzt einen eindeutigen Namen**
 - **besteht aus Packages, Klassen usw.**
 - **Bietet abgegrenzte Funktionalität**
 - **versteckt Implementierungsdetails**
 - **definiert sämtliche Abhängigkeiten**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



Einführung: Modularisierung des JDKs

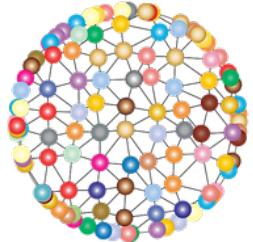


- **JDK-Verzeichnisstruktur massiv geändert, kein JDK / JRE mehr**
- **rt.jar und tools.jar existieren nicht mehr => Module in jmods**
- **Sichtbarkeitsbeschränkungen => einige interne APIs nicht mehr zugreifbar**
- **Erweiterung des JDKs mit »endorsed dirs« nicht mehr unterstützt**
- **Split Packages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



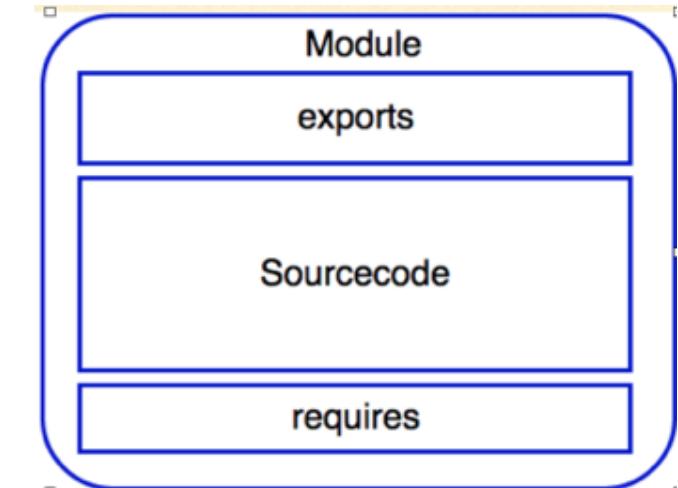


**Und wie sehen jetzt
Module aus?**



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

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



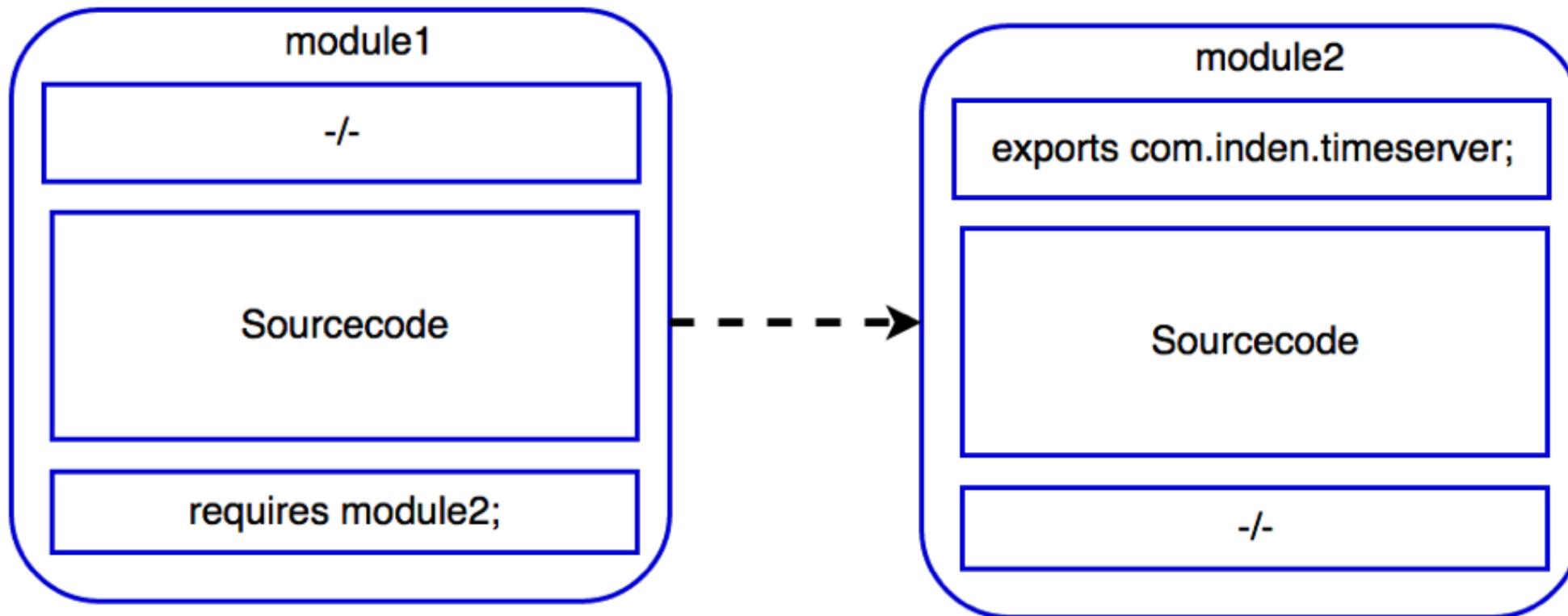
- **Enthält eine Menge von Packages und Klassen**
- **Definiert die Menge an Abhängigkeiten**
- **Das Modulsystem stellt sicher, dass jede Abhängigkeit genau durch ein Modul erfüllt wird und die Abhängigkeiten azyklisch sind.**

- **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 <modulepath> -m <modulename>/<moduleclass>
```



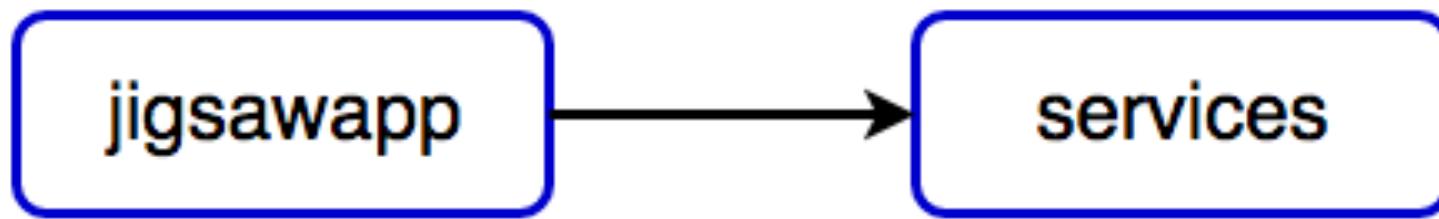
```
module module1
{
    requires module2;
}

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





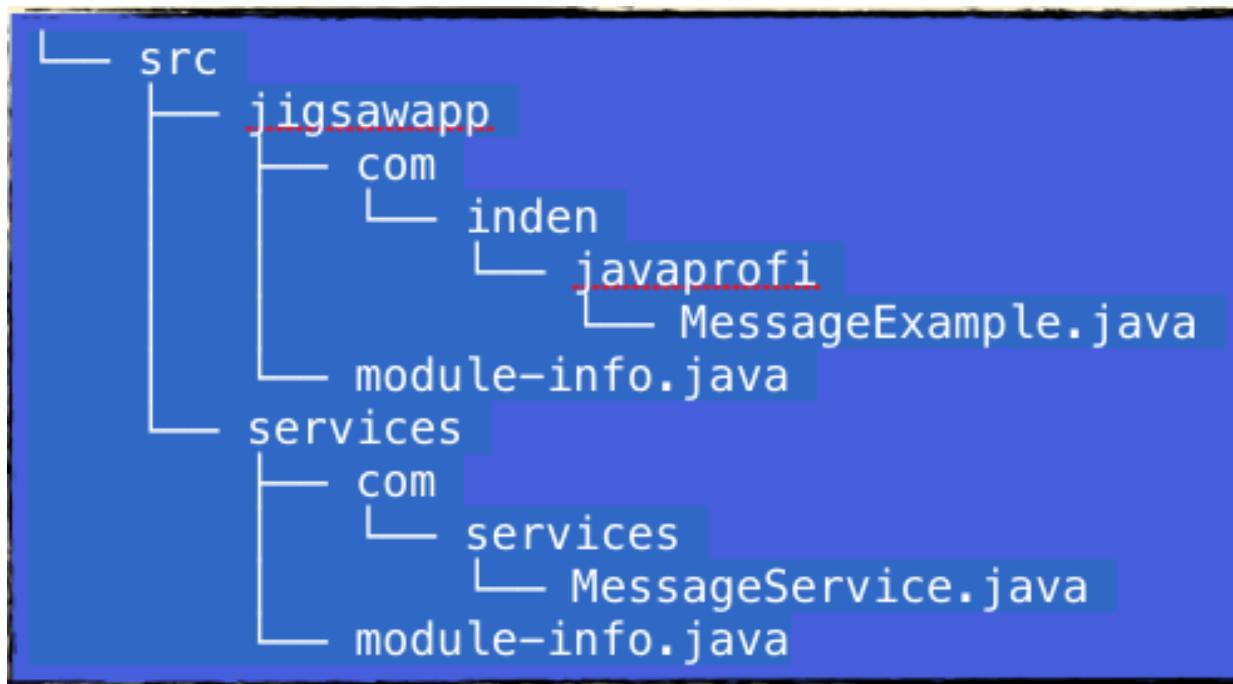
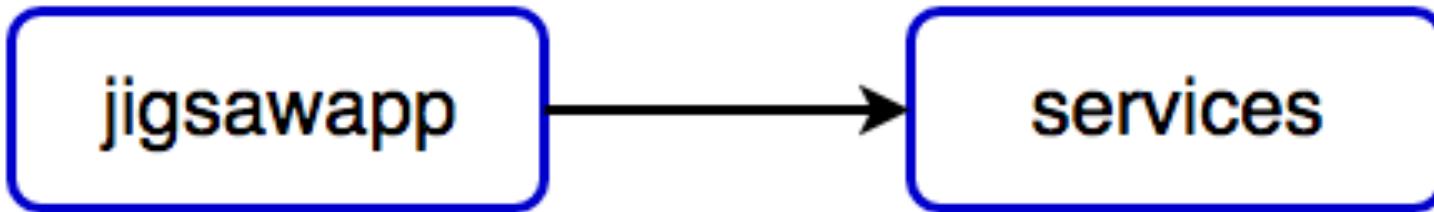
- Applikation mit mehreren Modulen => common src-Verzeichnis
- Pro Modul: Sourcecode in einem Unterverzeichnis mit dem Modulnamen abgelegt

```
'-- 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



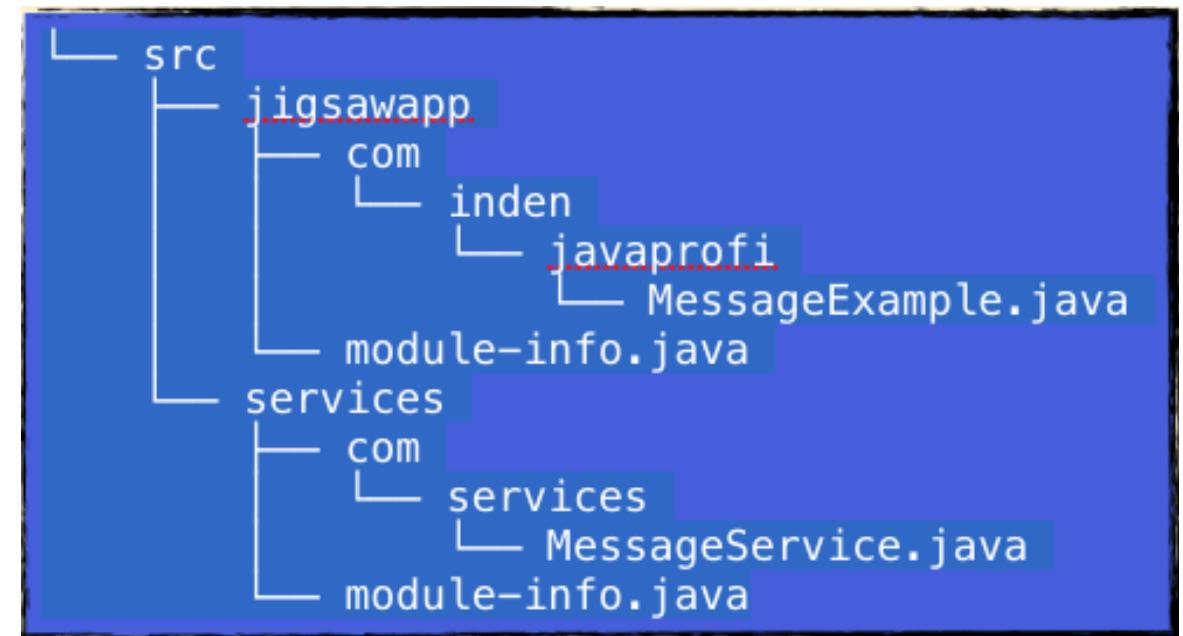
- **Schritt 1: Definition der Module**

- Strukturierung im Dateisystem:

```
mkdir -p src/jigsawapp  
mkdir -p src/services
```

- Moduldeskriptoren:

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



- **Schritt 2a: Anlegen der Verzeichnisse**

- mkdir -p src/services/com/services
- mkdir -p src/jigsawapp/com/inden/javaprofi

- **Schritt 2b: Implementierung der Klasse für das Modul services**

```
package com.services;

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

- Schritt 2: 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"));  
    }  
}
```

- **Schritt 3: Kompilieren erst Modul services, dann Modul jigsawapp**

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

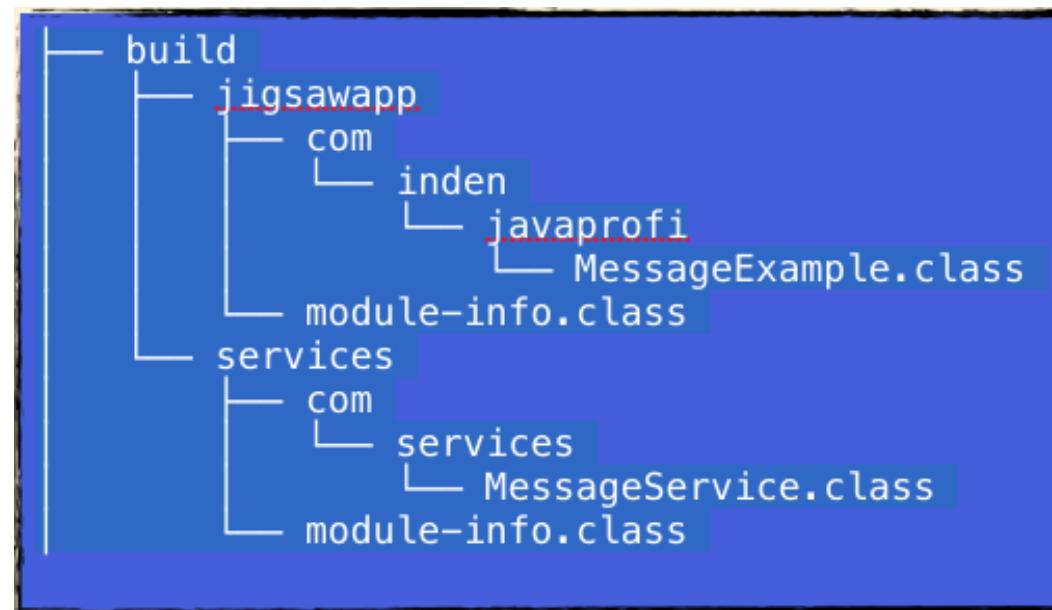
```
javac -d build/jigsawapp \
      src/jigsawapp/*.java \
      src/jigsawapp/com/inden/javaprofi/*.java
```

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

1 error

- **Schritt 3: Kompilieren des Moduls jigsawapp**

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



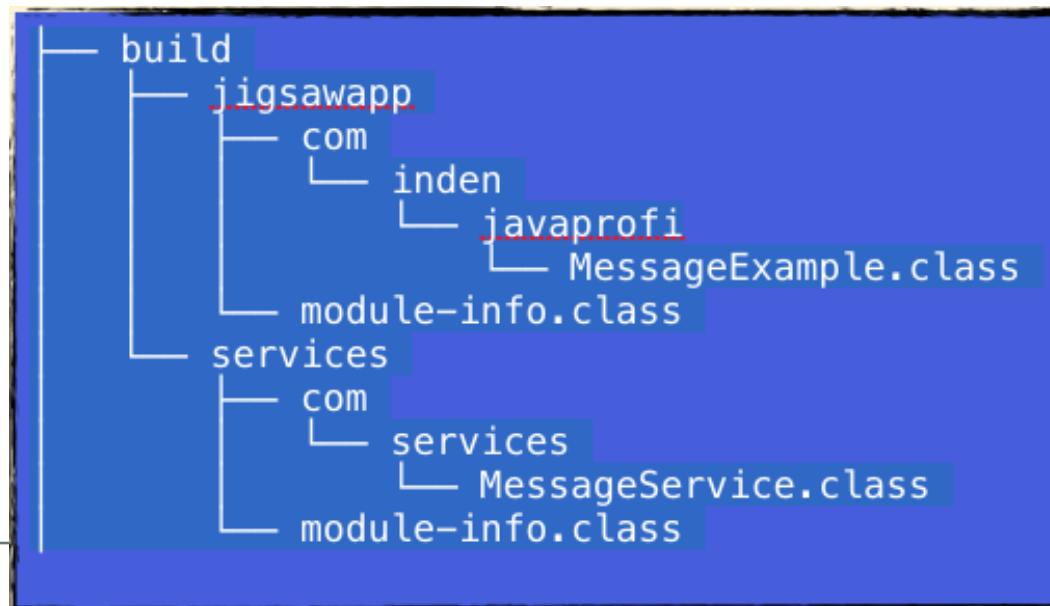
- **Schritt 3: Kompilieren mit Multi Module Build**

- Für MAC und Linux:

```
javac --module-source-path src -d build $(find src -name '*.java')
```

- Für Windows mit Powershell:

```
javac -d build --module-source-path src $(dir src -r -i '*.java')
```

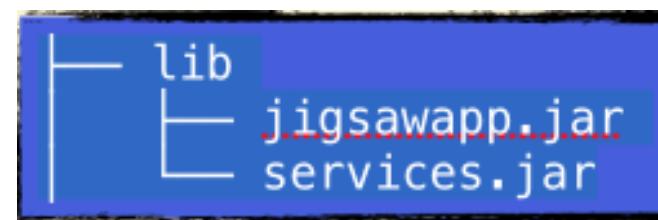


- **Schritt 4: Packaging**

```
mkdir lib
```

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

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



- **Schritt 5: Applikationsstart**

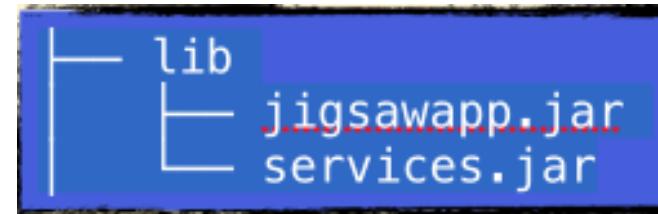
```
java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample
```

- **Schritt 4: 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 .
```

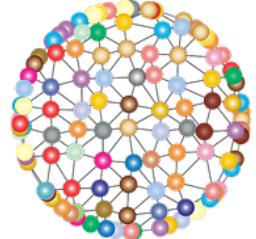


- **Schritt 5: Applikationsstart**

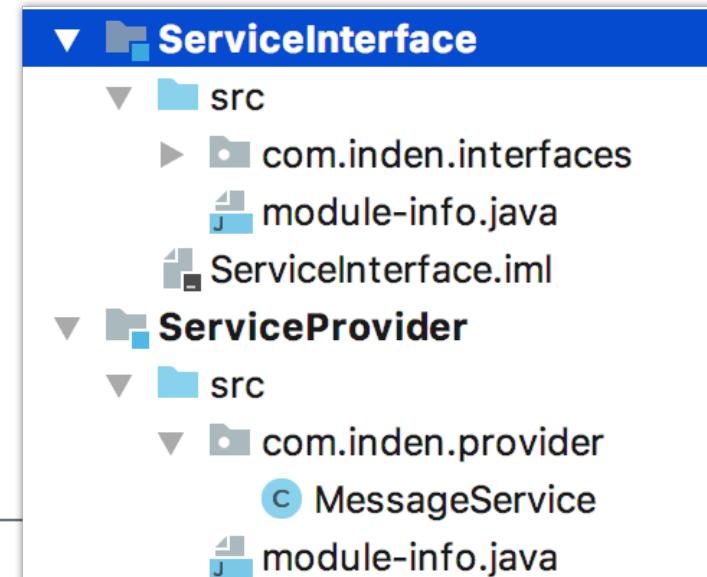
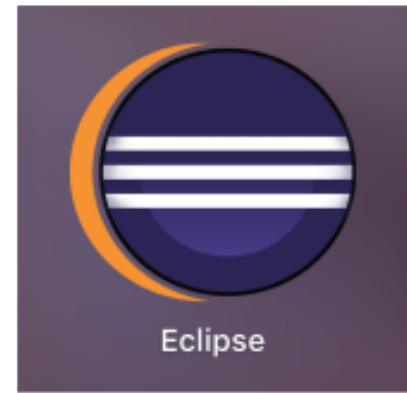
```
java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample  
java -p lib -m jigsawapp
```



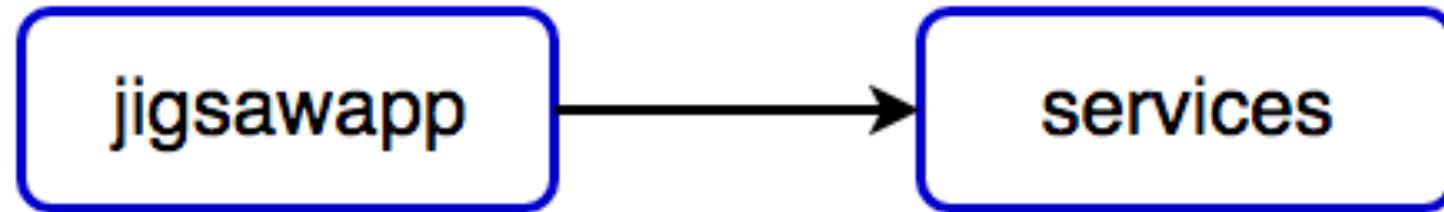
Module als High-Level-Bausteine, dann aber Low-Level-Konsolen-Junkie?



- 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



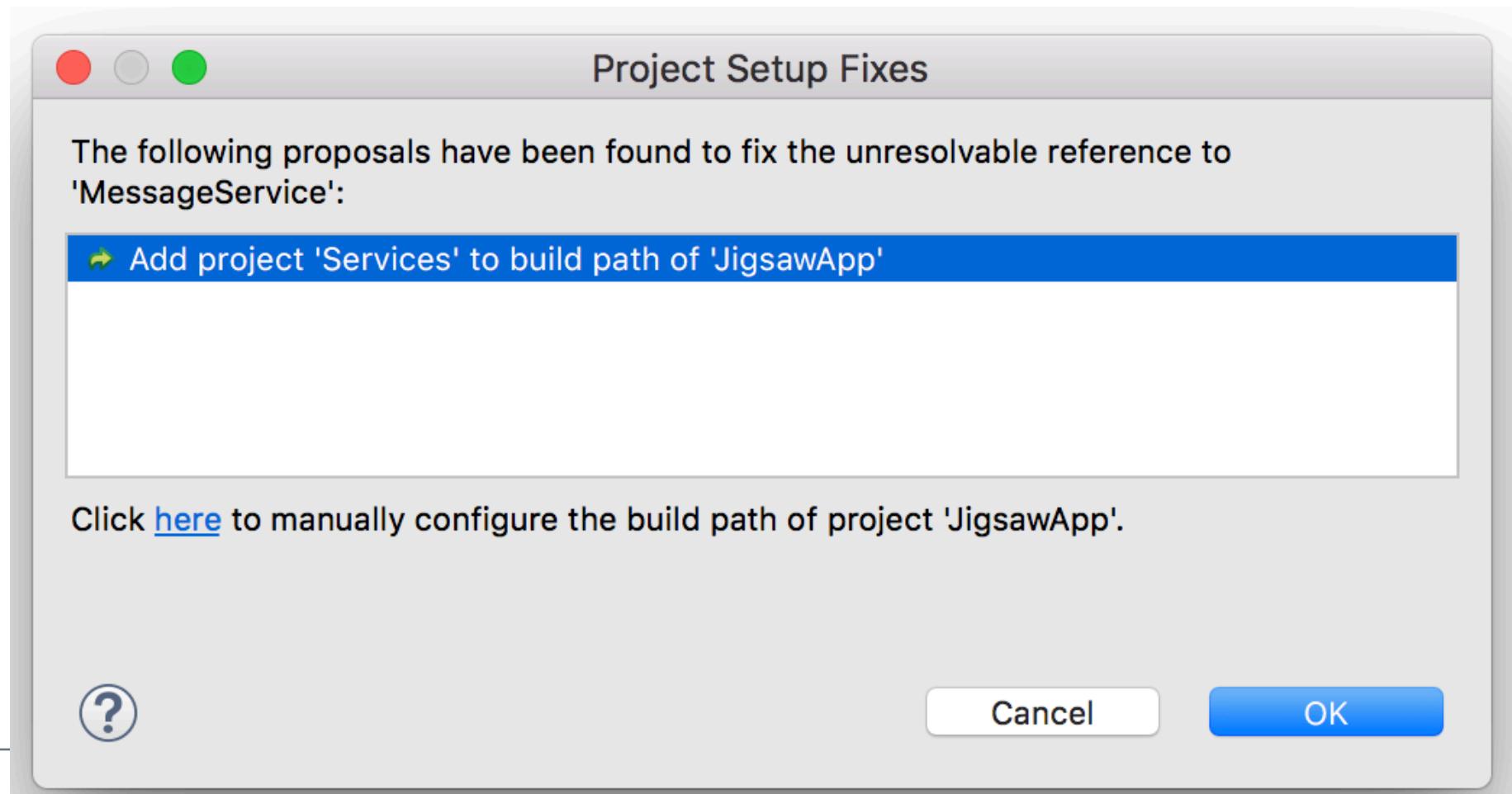
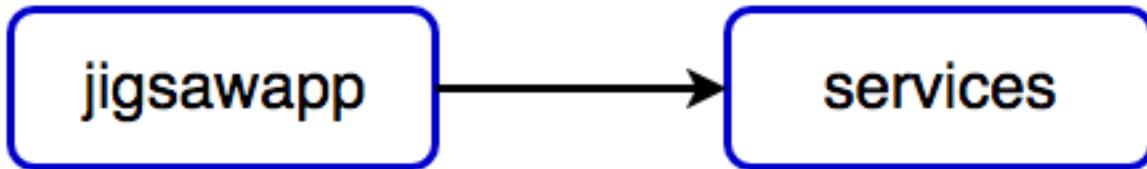
JigsawApp

- 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



- **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**
- ABER: Bei abweichendem Verzeichnislayout hat das JAR-Tool noch einen Fehler und wir können derart angelegte Projekte nicht zu korrekten modularen JARs zusammenbauen!!
Mit Java 9 war das noch möglich => Bugs-Request pending

Besonderheiten



- 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!

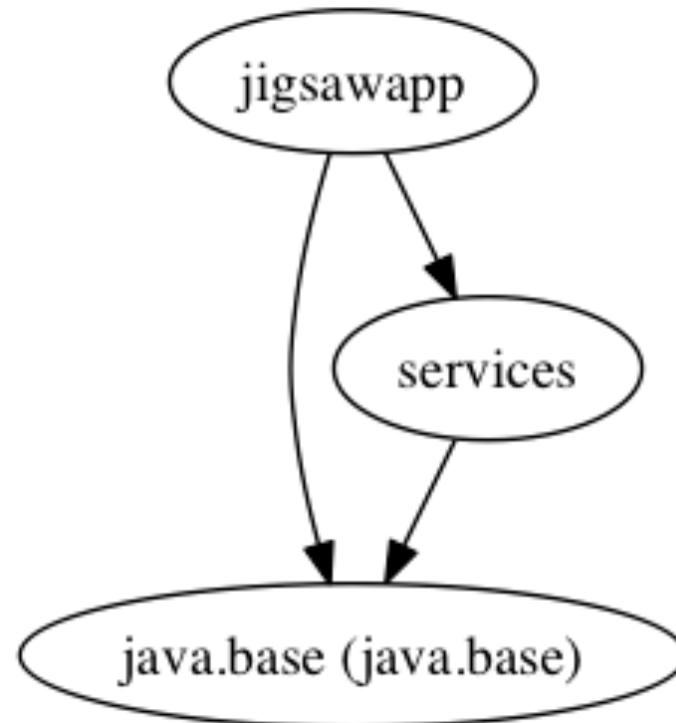
- Kompakte Aufbereitung der Abhängigkeiten

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

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

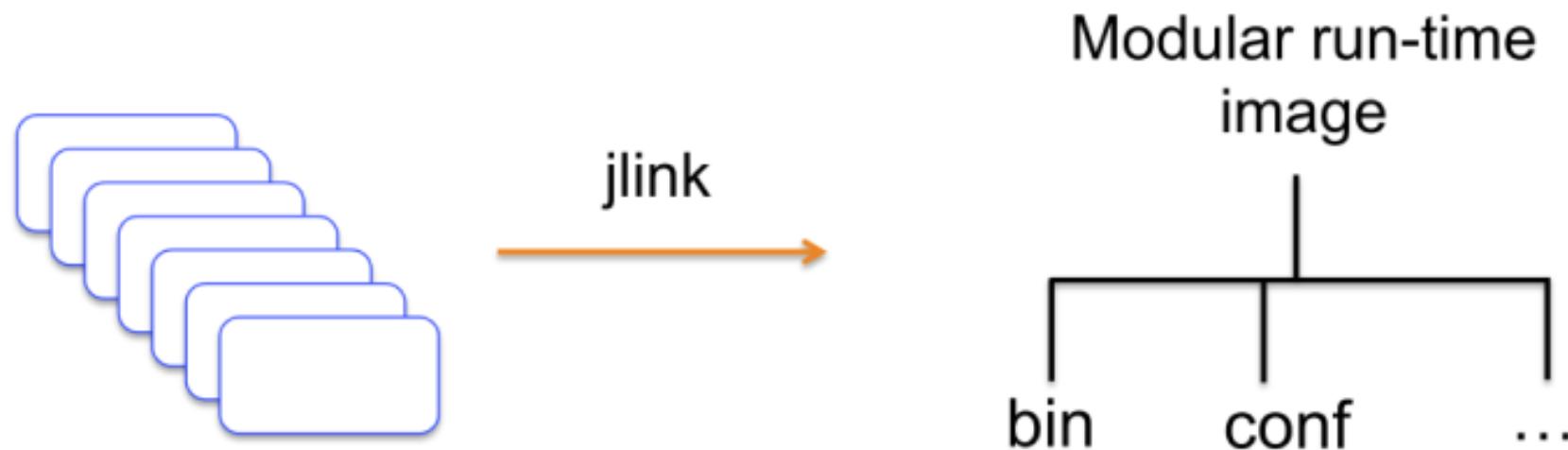
- Aufbereitung eines Abhängigkeitsgraphen

```
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

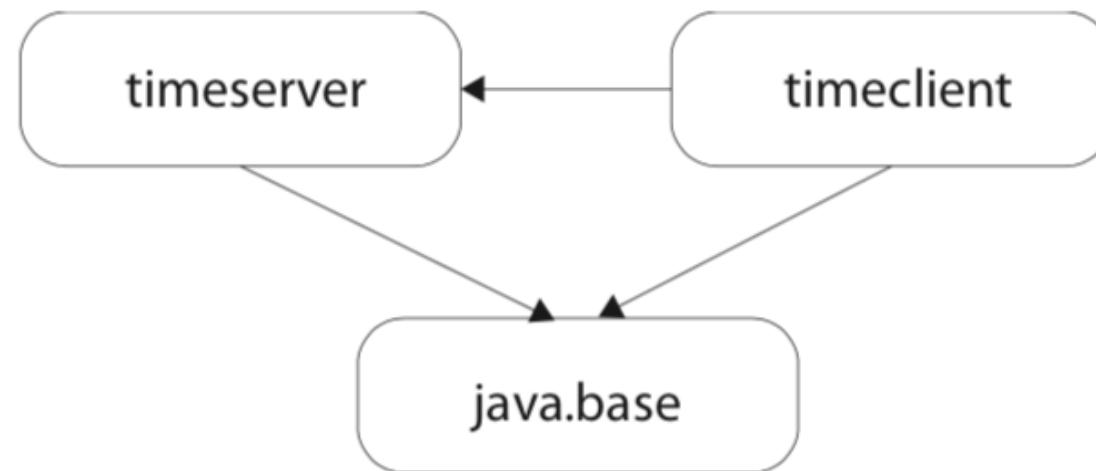
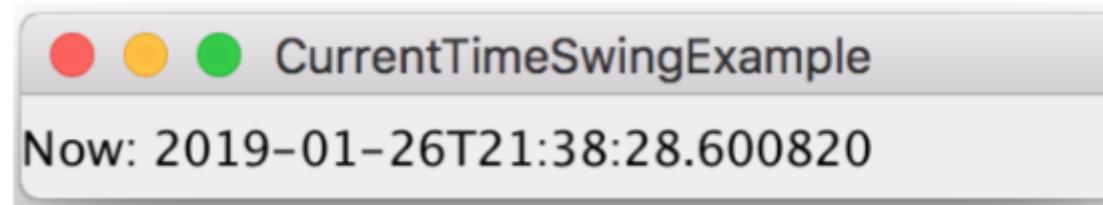
Zum Starten des Programms gibt man Folgendes ein:

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

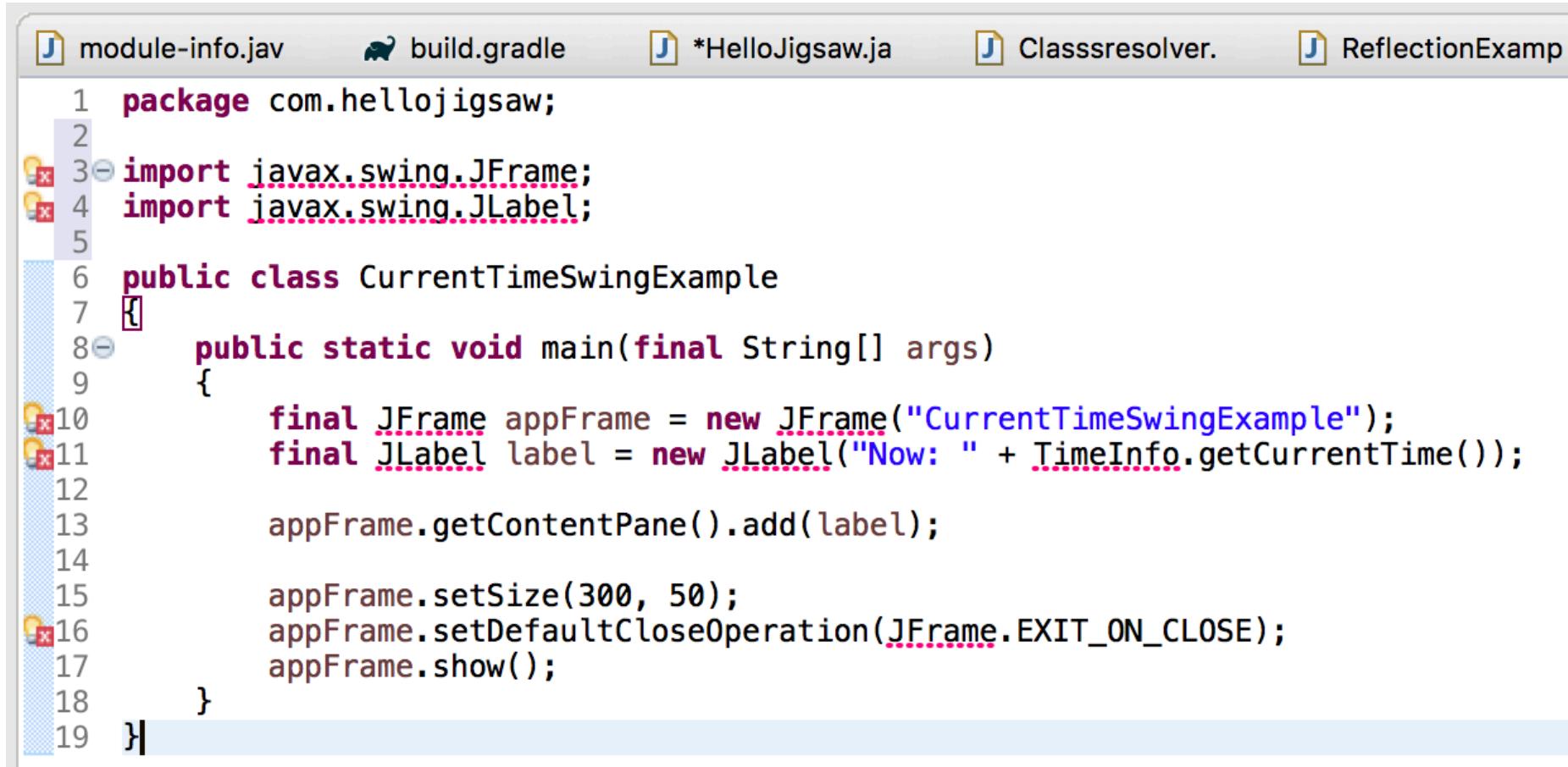
JDK-Module einbinden

- Bislang nur eigene Funktionalität genutzt => 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**

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



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



The screenshot shows a Java code editor with the following file structure:

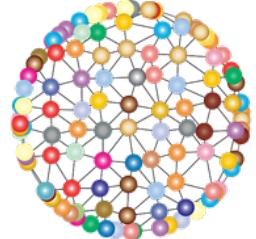
- module-info.java
- build.gradle
- *HelloJigsaw.java
- Classresolver.java
- ReflectionExample.java

The code in CurrentTimeSwingExample.java is as follows:

```
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?**



```
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.logging  
   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.logging  
   timeserver does not read it)  
src/timeclient/com/client/CurrentTime SwingExample.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/CurrentTime SwingExample.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/CurrentTime SwingExample.java:24: e  
    symbol  
    appFraem.pack();  
^  
symbol:  variable appFraem  
location: class CurrentTime SwingExample  
5 errors
```

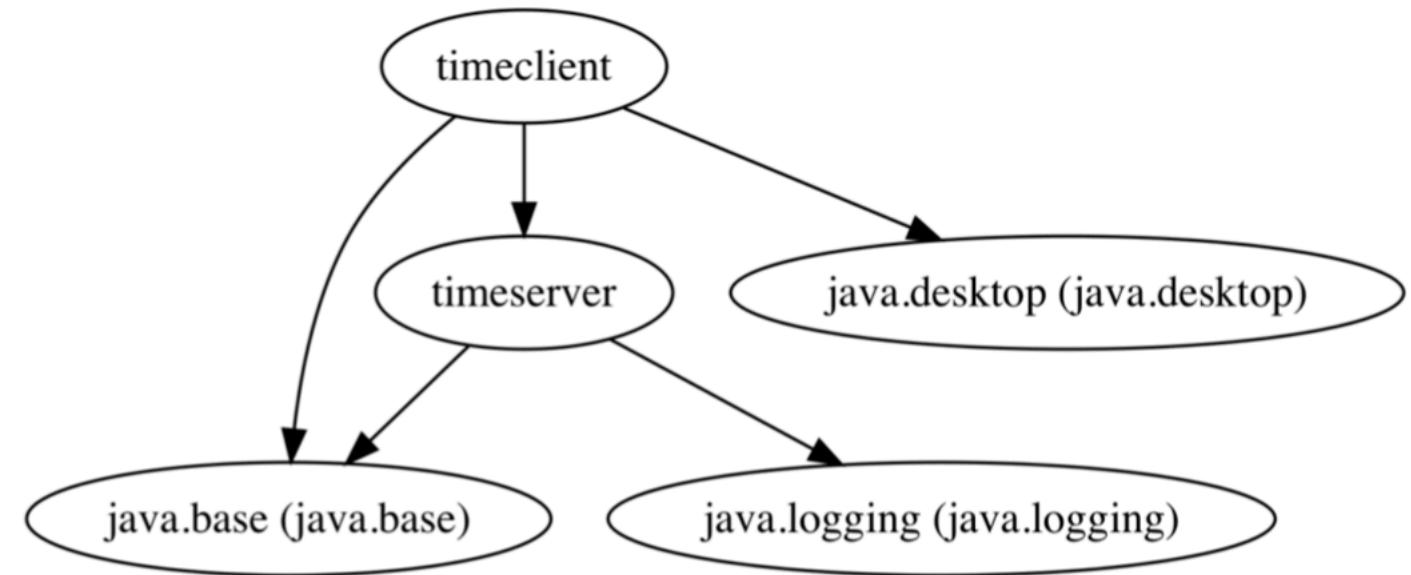


Press 'F2' for focus

Module des JDKs einbinden: Moduldeskriptoren anpassen

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

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



Anmerkungen zum Verzeichnislayout

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

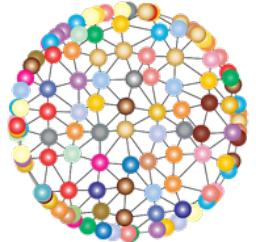
```
'-- src
  |-- com.inden.module1 <-- (highlighted)
  |   |-- com
  |   |   '-- inden
  |   |       '-- module1
  |   |           '-- Application.java
  |   '-- module-info.java
  '-- com.inden.module2 <-- (highlighted)
      |-- com
      |   '-- inden
      |       '-- module2
      |           '-- 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

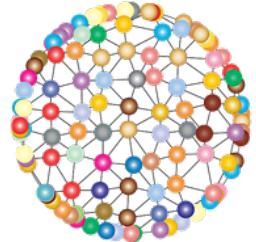
1. die Trennung der Module in verschiedene Projekte (für Build-Tools und IDEs),
2. das Erstellen von JARs als eigenständige Deployables sowie
3. die saubere Separation der Module untereinander.

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
```

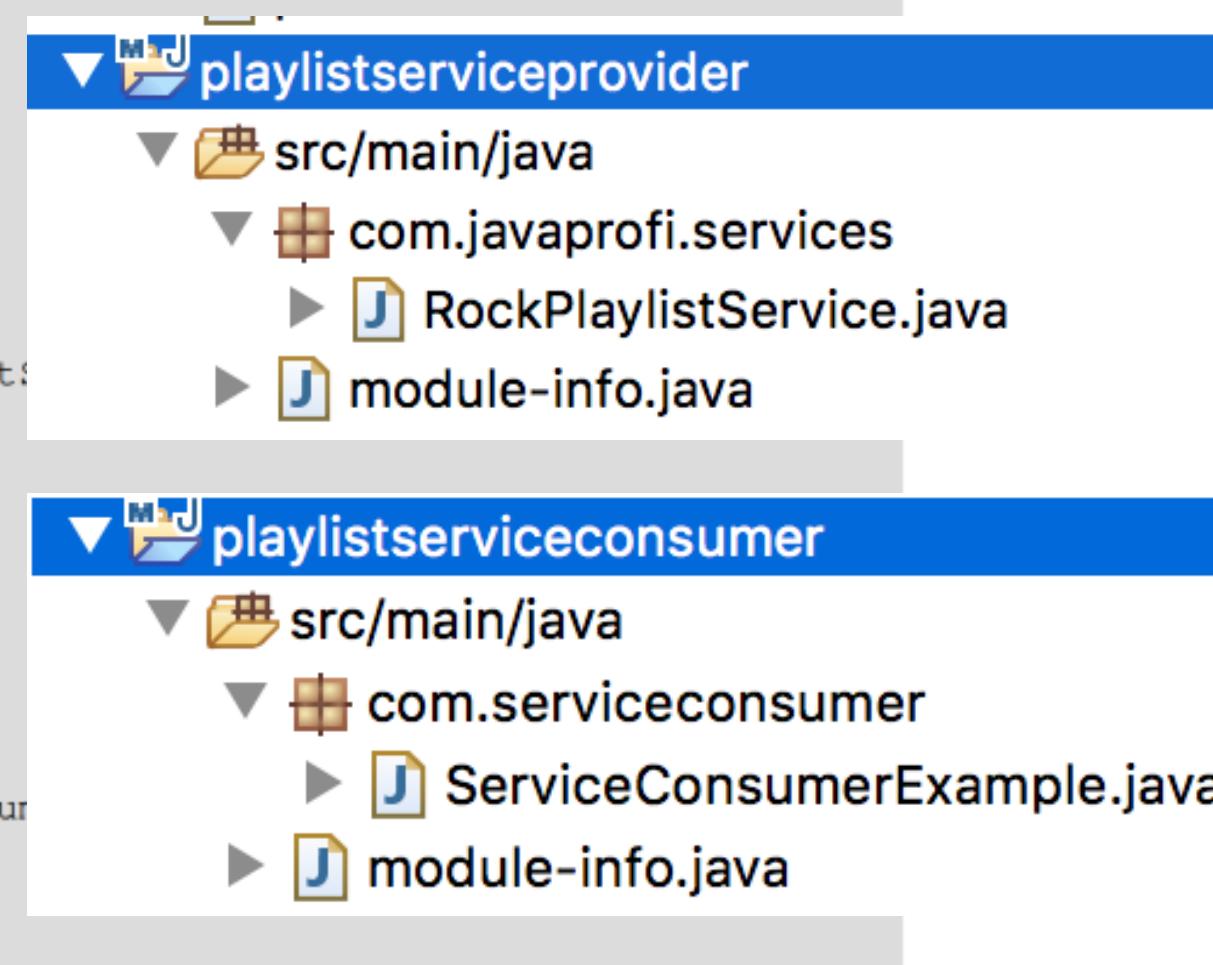


Variante 2: Applikation mit mehreren Modulen => mehrere *normale* src-Verzeichnisse

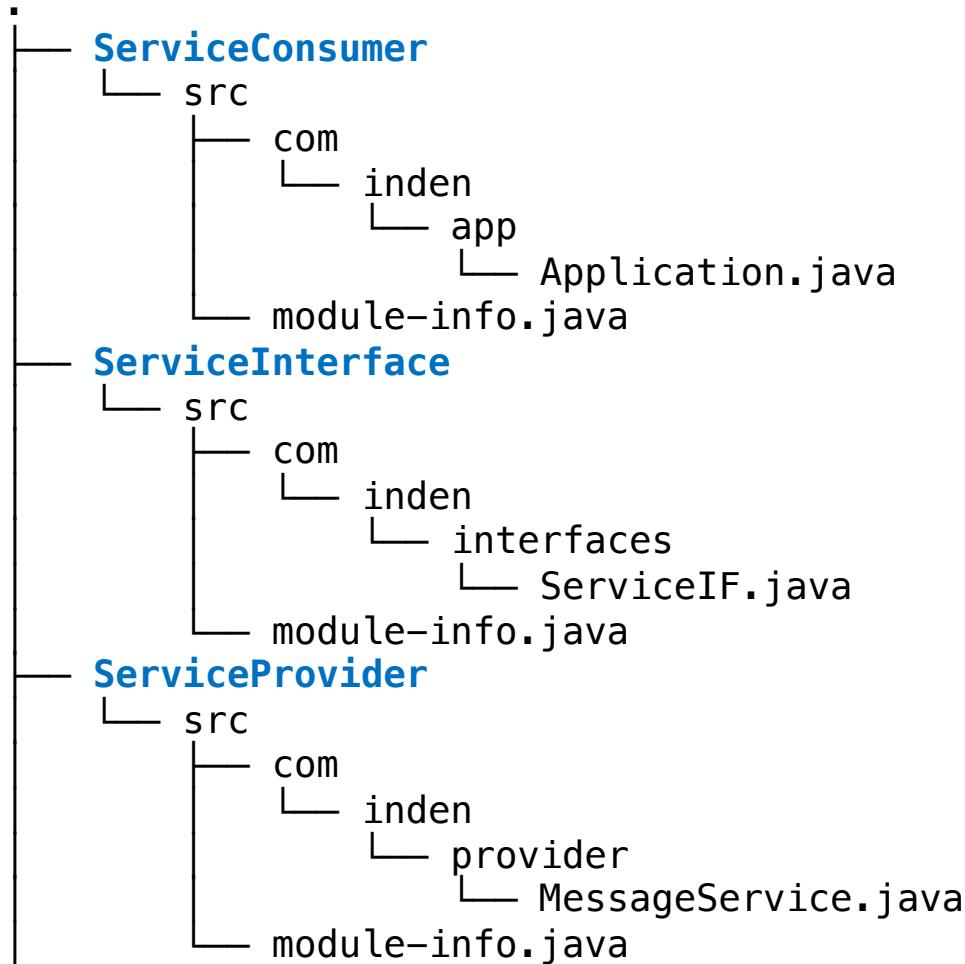
```
playlistservice-java9-modules-example
|--- playlistservice
|   |--- src
|   |   |--- main
|   |   |   |--- java
|   |   |   |   |--- playlistservice
|   |   |   |   |   |--- com
|   |   |   |   |   |   |--- javaprofi
|   |   |   |   |   |   |   |--- spi
|   |   |   |   |   |   |   |--- PlayListService.java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

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

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



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



- 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]  
}
```



Jigsaw Cheat Sheet Hands On

Übungen PART 1

Aufgabe 1 + 2 + 3 + 4

PART 2

Sichtbarkeiten und transitive Abhängigkeiten



Sichtbarkeiten

Sichtbarkeiten in JDK 8

Bis Java 8 besaßen Typen eine der folgenden vier Sichtbarkeiten:

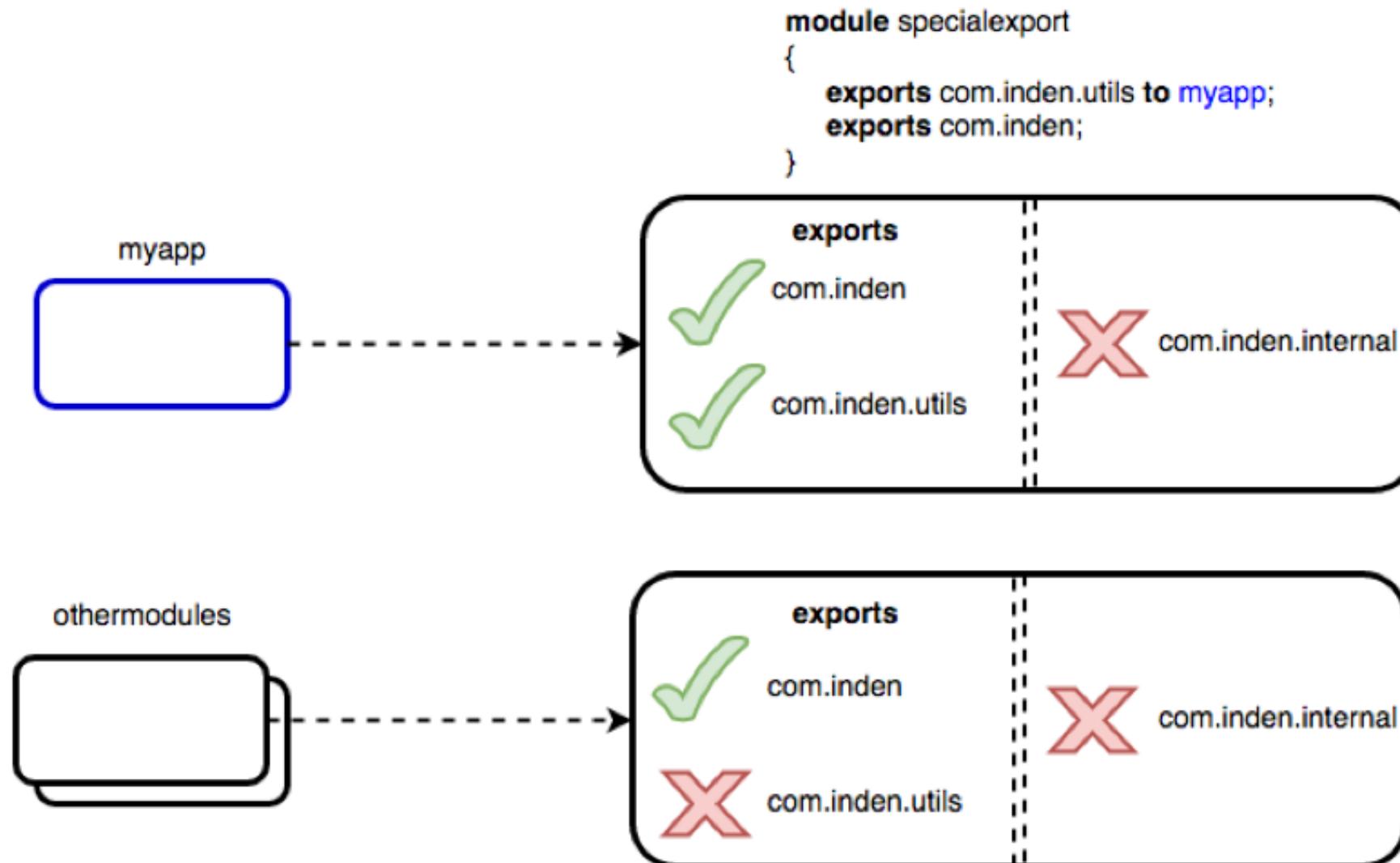
- `private` – Nur in der eigenen Klasse sichtbar
- `default / package private` (kein Schlüsselwort) – Nur im eigenen Package sichtbar
- `protected` – Wie `package private`, aber auch in abgeleiteten Klassen sichtbar
- `public` – Aus allen Packages zugreifbar

Tatsächlich bedeutet `public`, dass ein solcher Typ im CLASSPATH für alle anderen Typen zugänglich ist. Demnach kann man – zumindest für `public` – nicht wirklich von einer Sichtbarkeitssteuerung sprechen.

Sichtbarkeiten in JDK 9

Mit der Einführung der Modularisierung lässt sich die Sichtbarkeit von Typen genauer spezifizieren. Relevant ist dies in der Regel nur für die als `public` definierten Typen.

- **Global** — `public` für alle Module – Wenn das entsprechende Package einer `public` definierten Klasse mit `exports` zum Zugriff freigegeben wurde, ist diese von allen anderen Modulen zugreifbar, die auf dieses Modul per `requires` verweisen.
- **Eingeschränkt** auch *Qualified Export* genannt – `public` für angegebene Module – Mithilfe von `exports to` kann eine Liste von Modulen spezifiziert werden, die Zugriff erhalten sollen. Das erfordert zudem, dass andere Module auf dieses Modul per `requires` verweisen.
- **Modul intern** – `public` im Modul selbst – Sofern Klassen in Packages liegen, die nicht in `exports` aufgeführt werden, sind diese Klassen nur aus den Packages des eigenen Moduls zugreifbar.



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

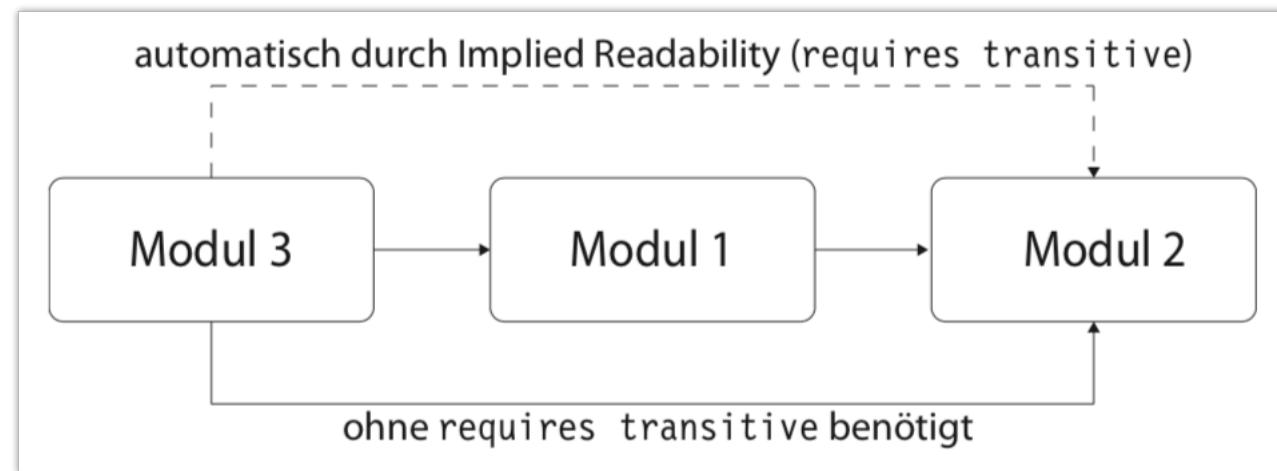
- in den Moduldeskriptoren nicht freigegebene Zugriffe werden verhindert.
- ein eingeschränkter Export für eine definierte Menge an Modulen möglich

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

Transitive Abhangigkeiten (Implied Readability)

Implied Readability

- Die in Moduldeskriptoren beschriebenen Abhängigkeiten werden nicht automatisch an nutzende Module propagiert.
- Immer dann, wenn mehrere Module kombiniert werden, kann dies umständlich werden:
- Modul 1 nutzt Modul 2, aber Modul 3 nutzt Modul 1
=> Modul 3 müsste dieses zusätzlich auch immer noch auf Modul 2 verweisen, um die Abhängigkeit zu erfüllen.



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;  
}
```

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

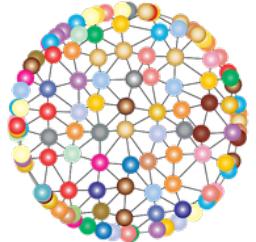


Module erlauben das **Untergliedern** einer Applikation in mehrere Bestandteile:

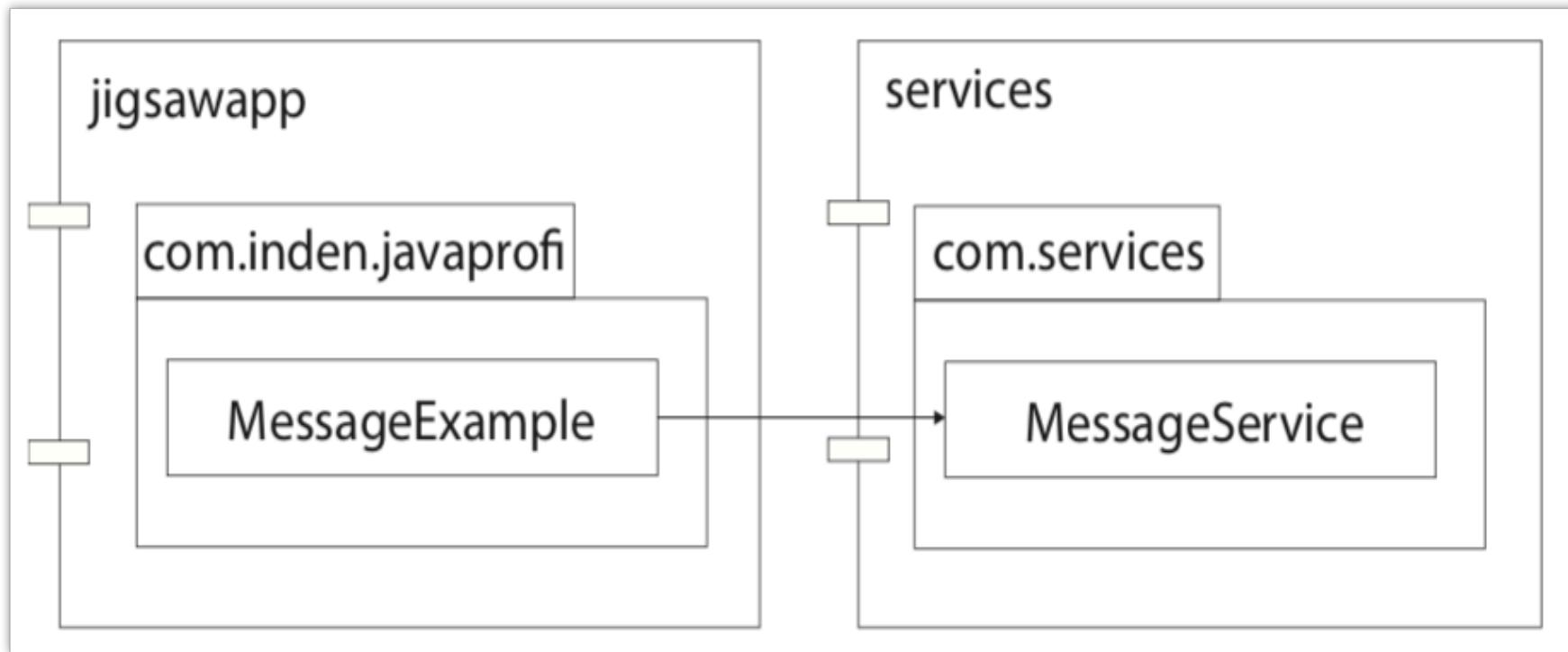
- **Aber bisher: Ein Modul verweist direkt auf ein anderes bzw. genauer eine Klasse nutzt eine Klasse eines anderen Moduls direkt.**
- **Unproblematisch für die Nutzung von Klassen des JDKs**
- **Mitunter fragwürdig bei eigenen Applikationen => stärkere Implementierungsabhängigkeit**



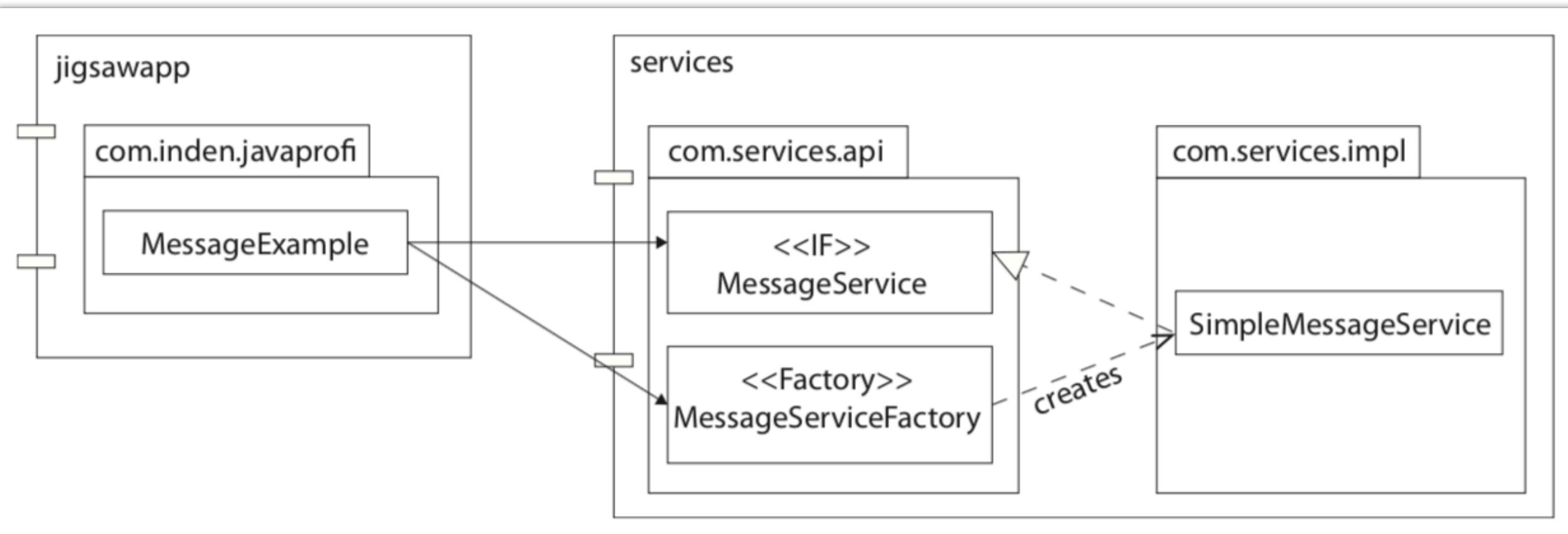
**Was ist daran
problematisch?**



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

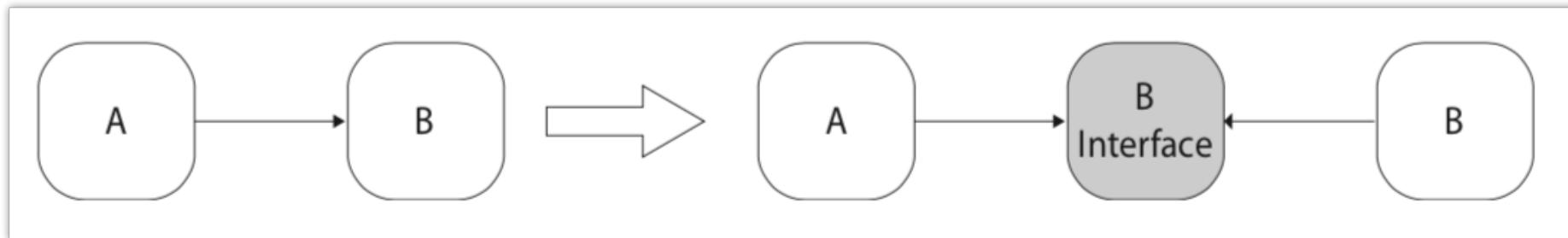


Soll eine stärkere Entkopplung erfolgen, so verweist eine nutzende Applikation idealerweise lediglich auf Interfaces und erhält die Realisierungen über **Factory-Methoden** oder **Services**.



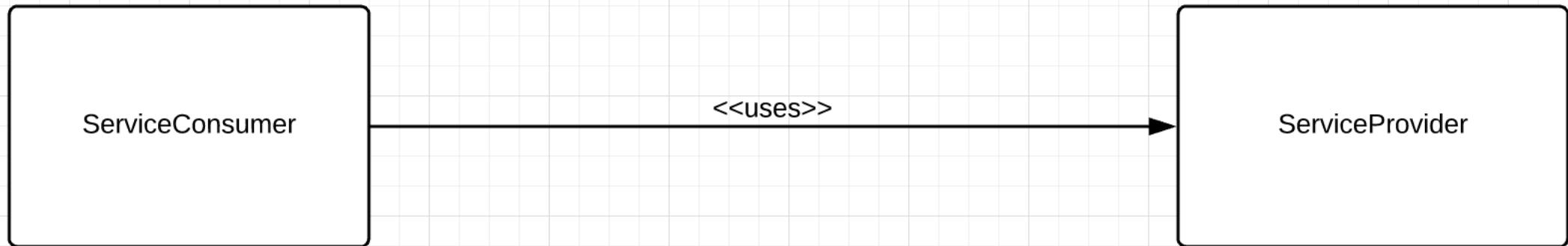
Was erreichen wir durch die Einführung der Indirektion und der Factory?

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](#).

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

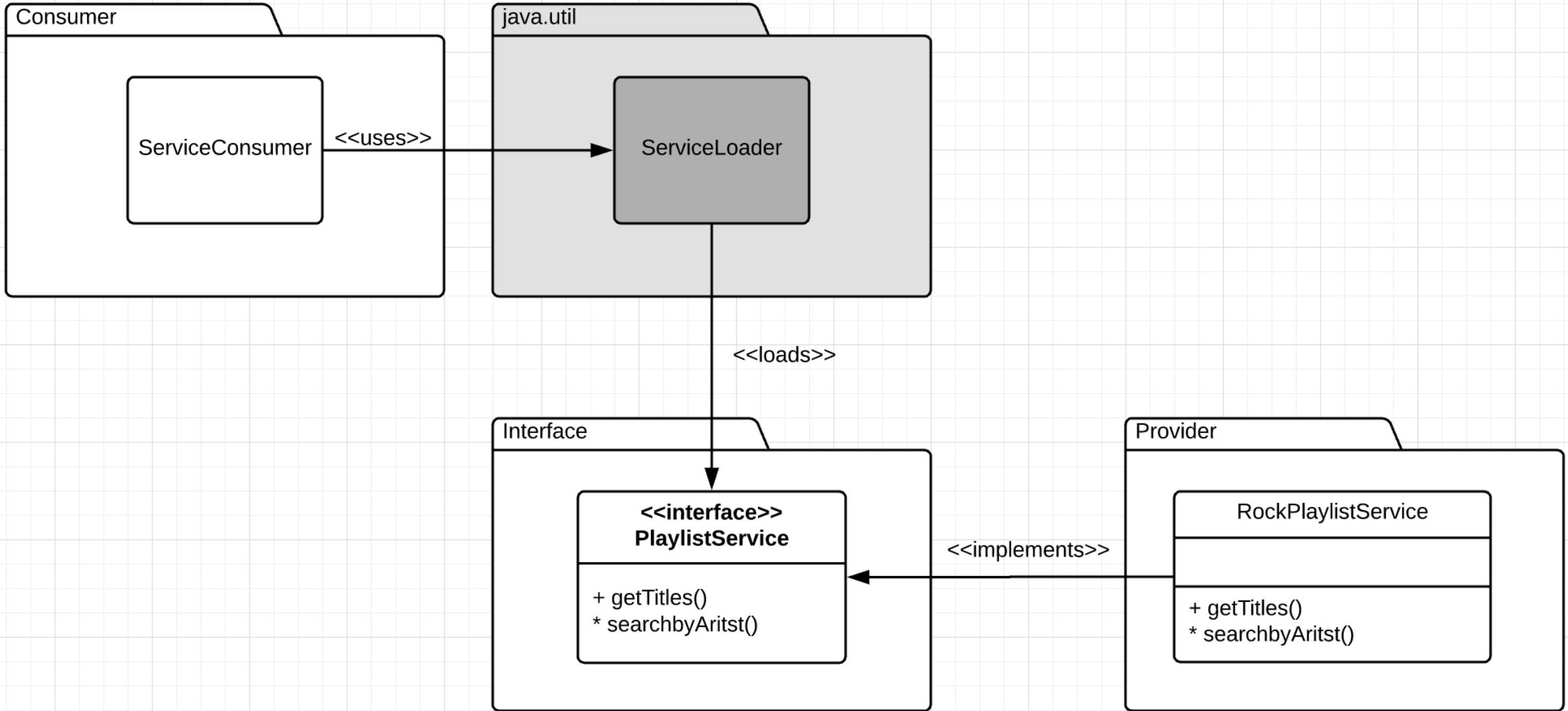
- Die Klasse **java.util.ServiceLoader**

- 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.

Services – Zu realisierende Applikation



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

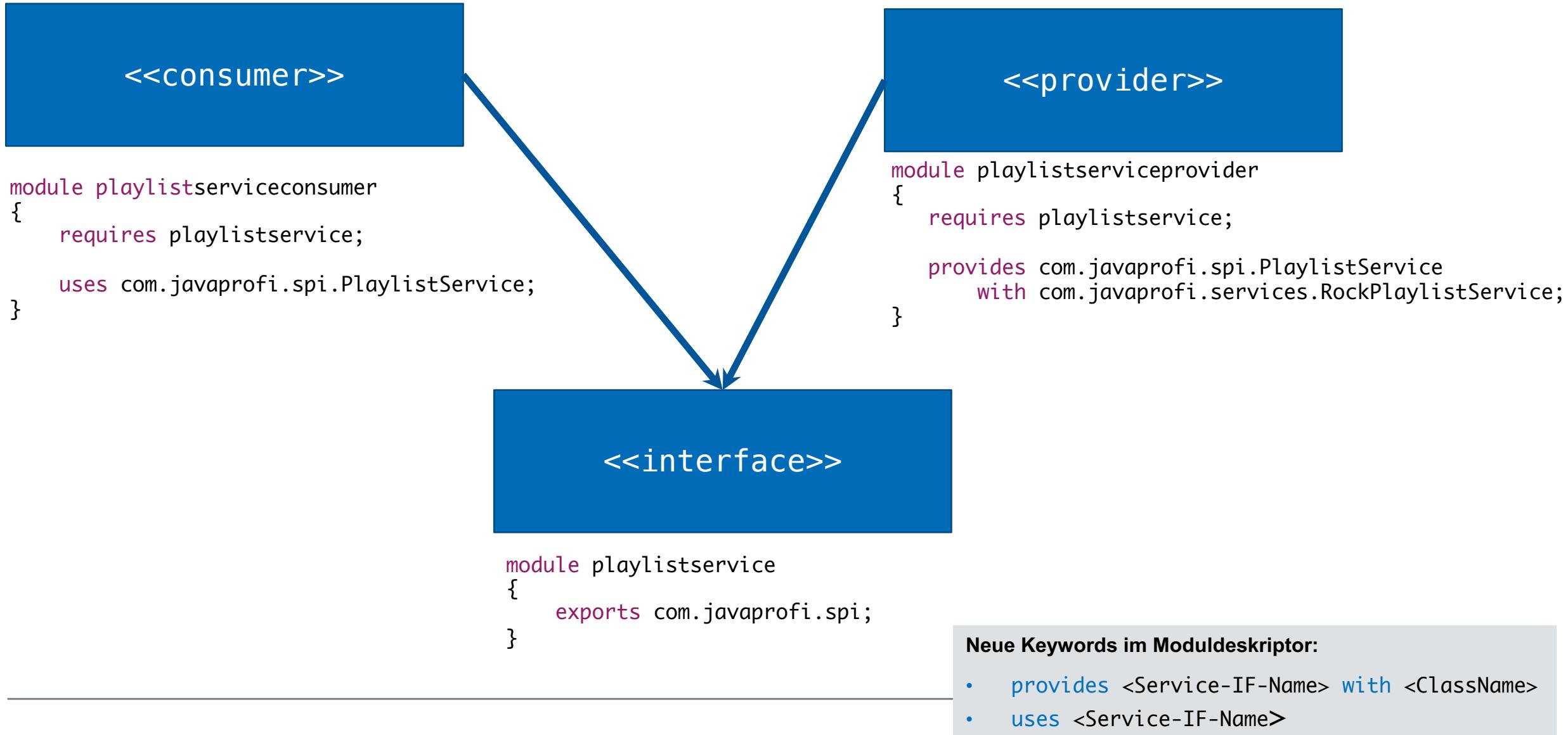
```
public static void main(final String[] args) throws Exception
{
    final Optional<PlaylistService> optService = lookup(PlaylistService.class);

    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();
}
```



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;  
}
```

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

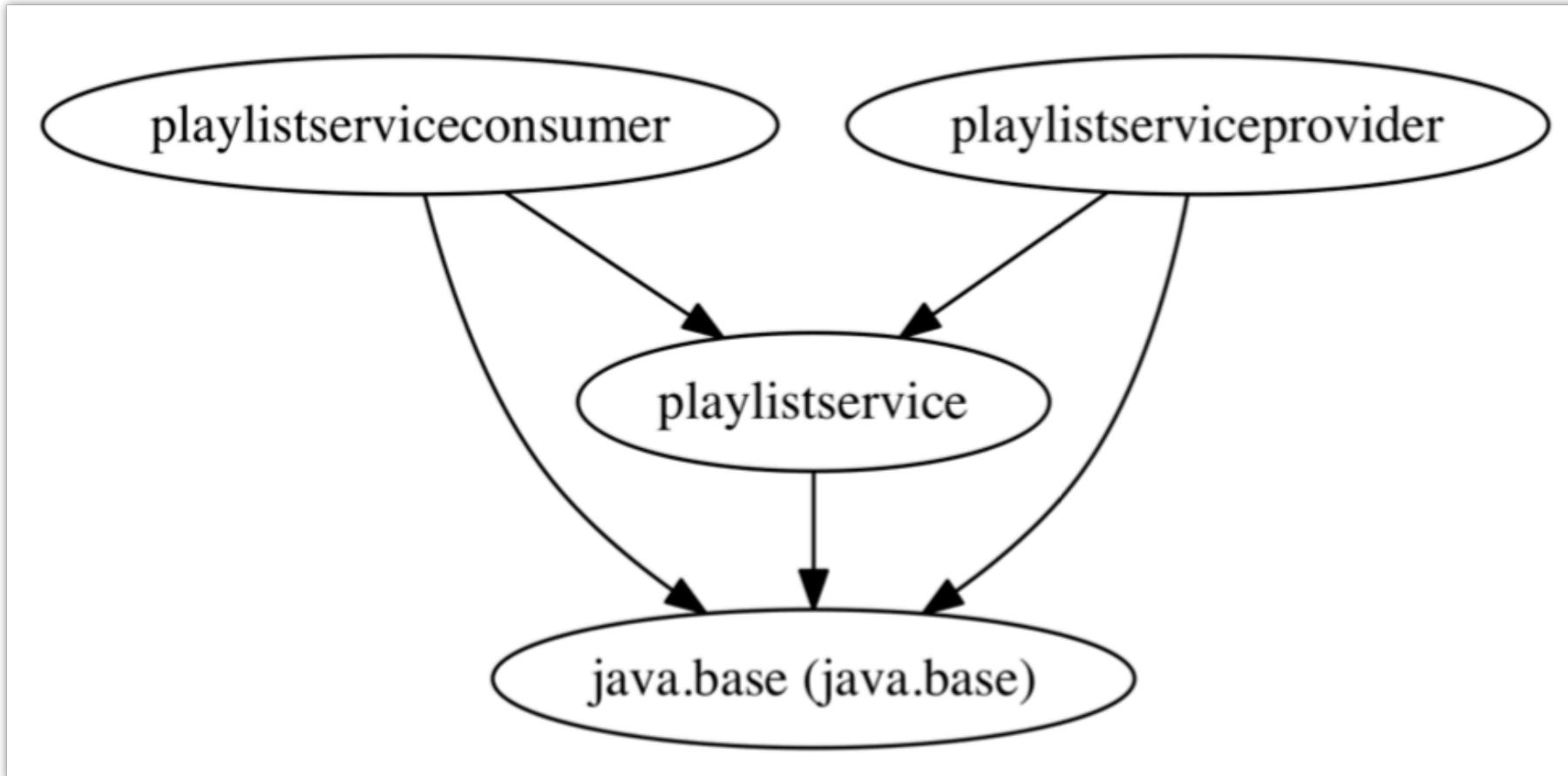
```
module playlistserviceprovider  
{  
    requires playlistservice;  
  
    provides com.javaprofi.spi.PlaylistService  
        with com.javaprofi.services.RockPlaylistService;  
}
```

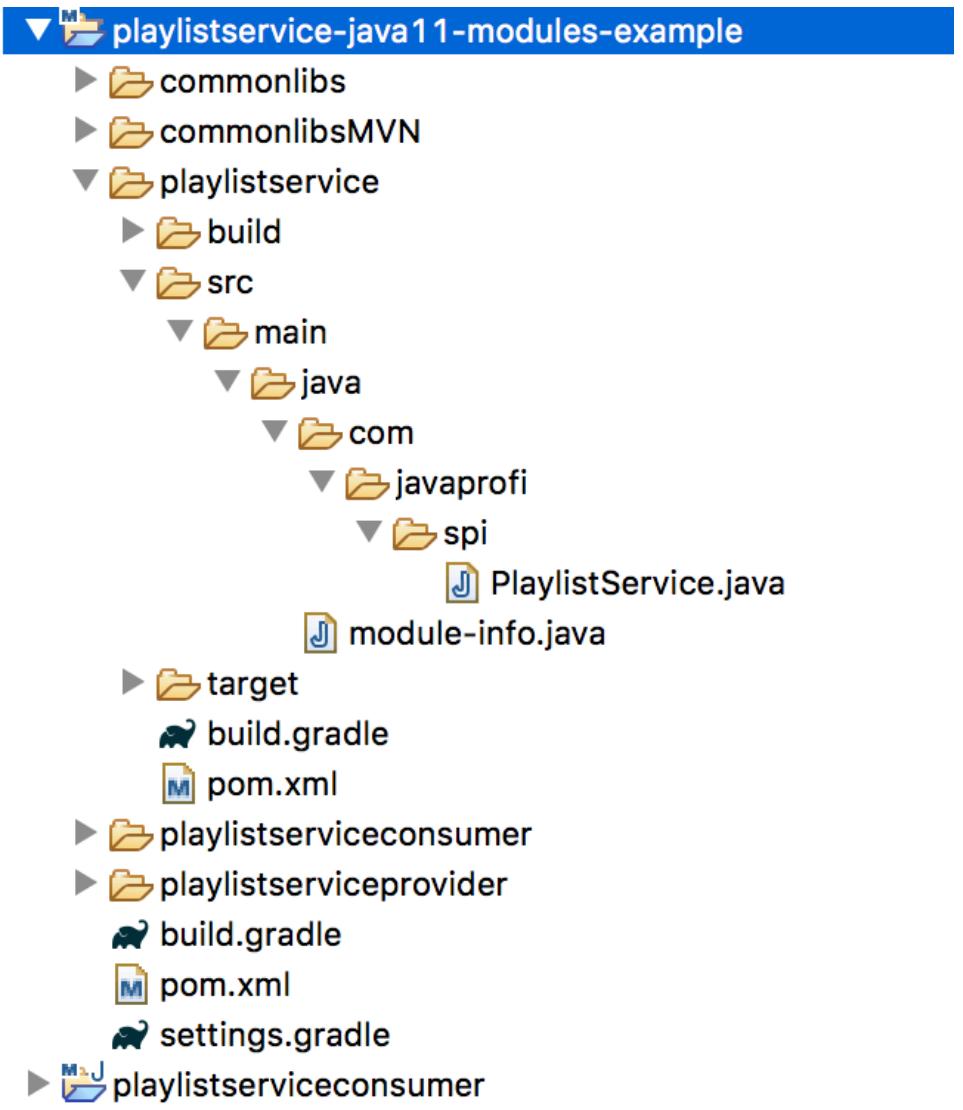
a) Implementierung des Service Consumers (analog zu vorher)

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

b) Definition des Moduldeskriptors

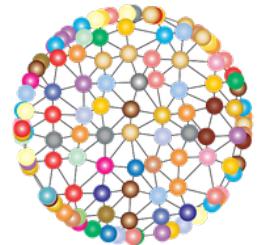
```
module playlistserviceconsumer  
{  
    requires playlistservice;  
  
    uses com.javaprof.spi.PlaylistService;  
}
```





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;
}
```

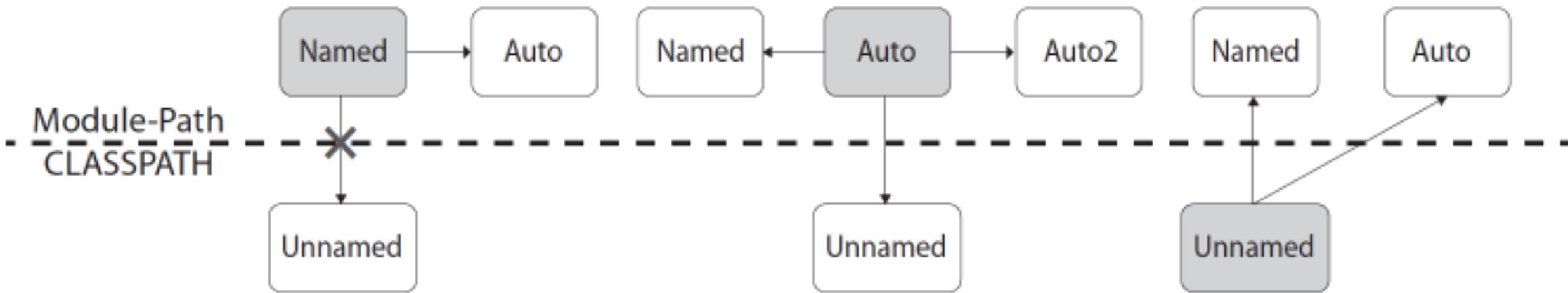
Exkurs Modularten

- **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.

Modultyp	Ursprung	Exporte	Readability
Platform	JDK	durch exports	-/-
Application	JAR mit Moduldeskriptor	durch exports	Platform, Application, Automatic
Automatic	JAR ohne Moduldeskriptor	alle Packages	Platform, Application, Automatic, Unnamed
Unnamed	alle JARs im CLASSPATH	alle Packages	Platform, Application, Automatic

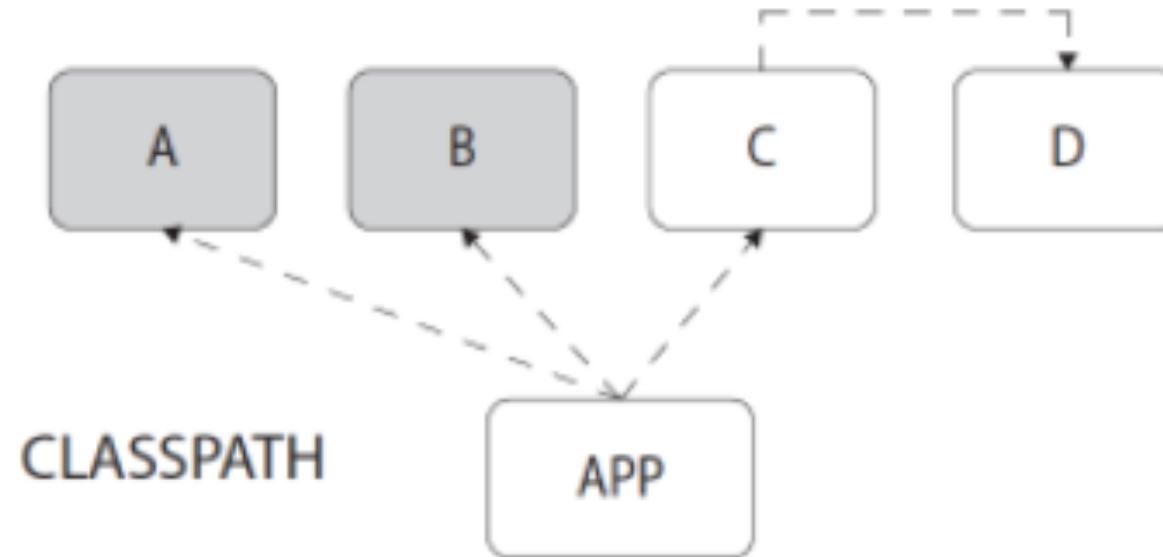
Zugriffsmöglichkeiten verschiedener Modulararten



Migrationen



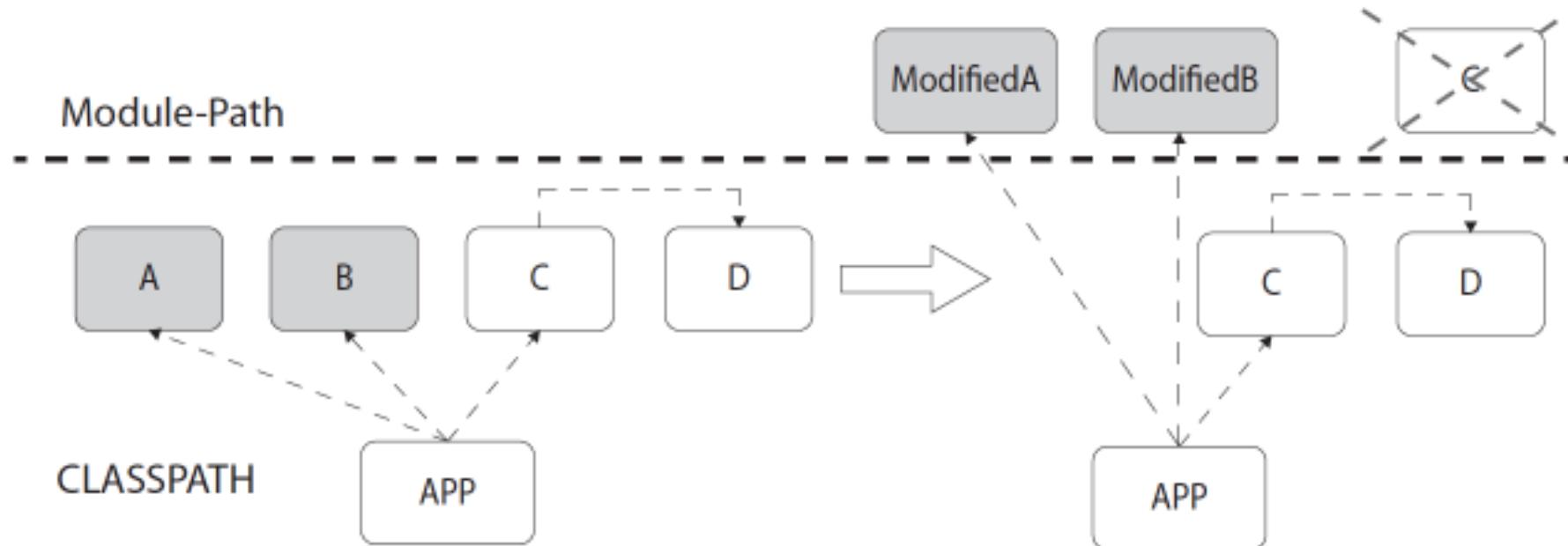
Betrachten wir eine Migration einer Applikation bestehend aus einigen JARs im CLASSPATH, etwa app.jar , A.jar , B.jar und C.jar:



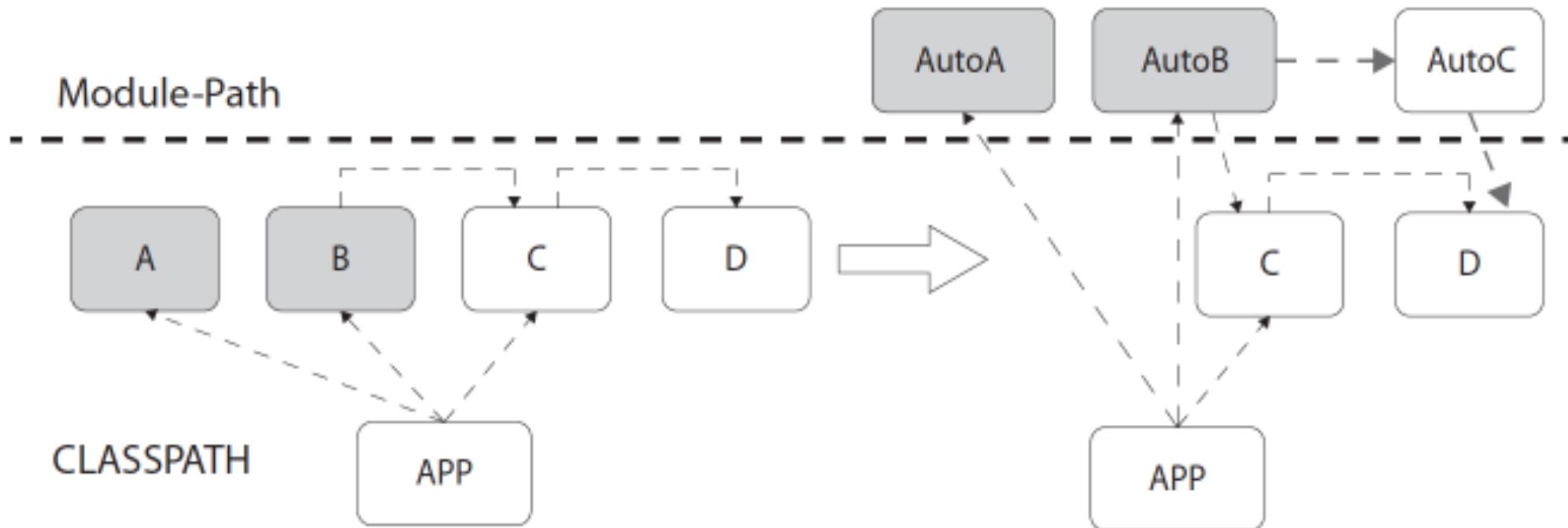
Bei einer sogenannten Bottom-up-Migration werden diese JARs schrittweise in modulare JARs umgewandelt. Dabei kann man folgende zwei Szenarien unterscheiden:

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.



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.



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!

Übungen PART 3 + 4

Aufgabe 5 + 6

Part 5

Jigsaw und Reflection

Introspection

- Mit einem **ModuleFinder** kann man **Module** aus **Verzeichnissen** oder **diejenigen des Systems, also des JDKs, ermitteln.**

```
final ModuleFinder finder = ModuleFinder.of(dir1, dir2, dir3);
final ModuleFinder systemFinder = ModuleFinder.ofSystem();
```

- Eine **ModuleReference** bietet Zugriff auf **Name** und **Verzeichnis** sowie den **Moduldeskriptor** eines Moduls.
- Die Klasse **ModuleDescriptor** modelliert einen **Moduldeskriptor**.
- Die Klasse **Module** repräsentiert die mit Project Jigsaw in das JDK neu eingeführten Module.

- Die Klasse **Module** repräsentiert die neu eingeführten Module.

```
public static void main(final String[] args) throws ClassNotFoundException
{
    final Optional<Module> optLogModule = ModuleLayer.boot().findModule("java.logging");
    optLogModule.ifPresent(logModule ->
    {
        final Optional<Module> optRmiModule = ModuleLayer.boot().findModule("java.rmi");
        optRmiModule.ifPresent(rmiModule -> printModuleInfo(logModule, rmiModule));
    });
}

private static void printModuleInfo(final Module logModule, final Module rmiModule)
{
    System.out.println("Module: " + logModule);
    System.out.println("Packages: " + logModule.get_packages());
    System.out.println("log canRead RMI: " + logModule.canRead(rmiModule));
    System.out.println("RMI canRead log: " + rmiModule.canRead(logModule));
}
```

- Die Klasse **Module** repräsentiert die neu eingeführten Module.

```
private static void printModuleInfo(final Module logModule, final Module rmiModule)
{
    System.out.println("Module: " + logModule);
    System.out.println("Packages: " + logModule.getPackages());
    System.out.println("log canRead RMI: " + logModule.canRead(rmiModule));
    System.out.println("RMI canRead log: " + rmiModule.canRead(logModule));
}
```

=>

```
Module: module java.logging
Packages: [sun.util.logging.internal, java.util.logging,
           sun.util.logging.resources, sun.net.www.protocol.http.logging]
log canRead RMI: false
RMI canRead log: true
```

- Die Klasse **ModuleDescriptor** modelliert einen Moduldeskriptor:

```
private static void printModuleDescriptorInfo(final ModuleDescriptor descriptor)
{
    System.out.println("name:      " + descriptor.name());
    System.out.println("requires:   " + descriptor.requires());
    System.out.println("provides:   " + descriptor.provides());
    System.out.println("exports:    " + descriptor.exports());
    System.out.println("automatic:  " + descriptor.isAutomatic());
    System.out.println("packages:   " + descriptor.packages());
}
```

- Für das Modul **java.logging**:

```
name:      java.logging
requires:  [mandated java.base]
provides:  [jdk.internal.logger.DefaultLoggerFinder with
[sun.util.logging.internal.LoggingProviderImpl]]
exports:   [java.util.logging]
automatic: false
packages:  [sun.util.logging.resources, sun.net.www.protocol.http.logging,
sun.util.logging.internal, java.util.logging]
```

```
private static Optional<ResolvedModule> findModuleByClassName(final String className)
{
    final Configuration config = ModuleLayer.boot().configuration();
    final Set<ResolvedModule> modules = config.modules();
    return modules.stream().filter(module -> tryFindClass(module, className)).findFirst();
}

public static boolean tryFindClass(final ResolvedModule resModule, final String name)
{
    final Optional<Module> optModule = ModuleLayer.boot().findModule(resModule.name());
    if (optModule.isPresent())
    {
        return Class.forName(optModule.get(), name) != null;
    }
    return false;
}
```

```
private static Optional<ResolvedModule> findModuleByClassName(final String className)
{
    final Configuration config = ModuleLayer.boot().configuration();
    final Set<ResolvedModule> modules = config.modules();
    return modules.stream().filter(module -> tryFindClass(module, className)).findFirst();
}
```

```
Optional<ResolvedModule> optModule = findModuleByClassName("javafx.application.Application");
Optional<ResolvedModule> optModule2 = findModuleByClassName("java.time.LocalDate");
```

```
// Ab Java 11
Optional.empty()
Optional[670700378/java.base]
```

```
// Bis Java 10
Optional[1775282465/javafx.graphics]
Optional[1775282465/java.base]
```

Zugriffe per Reflection

```
public static void main(final String[] args) throws Exception
{
    final Method method = URL.class.getDeclaredMethod("getURLStreamHandler",
                                                       String.class);
    method.setAccessible(true);
    System.out.println(method.invoke(null, "http"));
}
```

Die in die JVM integrierte Zugriffskontrolle produziert folgende Warnmeldung:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by ch11_3_3.ReflectionExample
(file:/Users/michael.inden/Desktop/JavaBooks/Java-Aktuell-JDK9-10-11-12-Die-
Neuerungen/quelltext/target/classes/) to method
java.net.URL.getURLStreamHandler(java.lang.String)
WARNING: Please consider reporting this to the maintainers of ch11_3_3.ReflectionExample
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access
operations
WARNING: All illegal access operations will be denied in a future release
sun.net.www.protocol.http.Handler@4c75cab9
```

- Die in die JVM integrierte Zugriffskontrolle verhindert bei der Modularisierung, dass den Zugriff per Reflection auf Elemente (Typen, Methoden, Attribute), die nicht public sind.
- Dies sichert den Zugriffsschutz, der ja einer der Vorzüge der Modularisierung ist.
- Für einige Anwendungsfälle doch immer einmal wieder wünschenswert, Reflection zuzulassen. Das gilt im Speziellen für einige Tools und Bibliotheken, etwa für Spring (für Dependency Injection) oder für JPA-Provider wie Hibernate (zur Persistierung).

- **Open Modules** – Module können durch das Schlüsselwort `open` vor `module` im Moduldeskriptor explizit und vollständig für Reflection geöffnet werden.
- **Das Schlüsselwort `opens`** – Im Moduldeskriptor können nach `opens` aufgezählte Packages explizit für den externen Zugriff per Reflection geöffnet werden. Dazu gibt es auch die Variante spezifischer Öffnung für bestimmte Module.
- **Kommandozeilenparameter `--add-opens`** – Mithilfe des Kommandozeilenparameters `--add-opens` können bei der Ausführung einzelne Packages explizit für Reflection geöffnet werden.

Open Modules

```
open module reflectionuser
{
    requires reflectionutils;
}
```

Schlüsselwort opens

```
module reflectionuser
{
    requires reflectionutils;

    opens com.domain; // Zugriff nur zur Laufzeit
}
```

Kommandozeilenparameter --add-opens

```
java -p build --add-opens reflectionuser/com.domain=reflectionutils \
      -m reflectionuser/com.reflectionuser.ReflectionUtilsUsageExample
```

- Alle drei «opens»-Varianten ändern an der Sichtbarkeit während des Kompilieren nichts
- Interessant ist der Einfluss zur Laufzeit

Schlüsselwörter	Kompilieren	Reflection	
		Shallow	Deep
exports	erlaubt	erlaubt	-/-
opens	-/-	erlaubt	erlaubt
exports und opens	erlaubt	erlaubt	erlaubt

Hinweis: Shallow Reflection und Deep Reflection

Bezüglich Zugriffsschutz und Reflection gab es einiges an Klärungsbedarf und unterschiedlichste Lösungsansätze. Schließlich wurde der starken Kapselung viel Wert beigemessen, was aber gerade auch für Build-Tools, Frameworks usw. durchaus das eine oder andere Problem verursacht hat. Reflection wird nun in die zwei Typen Shallow und Deep Reflection untergliedert:

- Reflection oder **Shallow Reflection** – Shallow Reflection erlaubt Zugriffe per Reflection, wenn dies auch direkt statisch möglich ist.
- **Deep Reflection** – Deep Reflection verkörpert Zugriffe, wie sie früher nur mit `setAccessible(true)` auf Typen stattfinden konnten, etwa weil die Sichtbarkeit geringer als `public` ist. Ohne Deep Reflection führt ein Aufruf von `setAccessible(true)` – wie eingangs im Beispiel gesehen – zu einer `java.lang.reflect.InaccessibleObjectException`.

Um mögliche Probleme zu adressieren, wurde mit dem Öffnen von Packages und Modulen eine Variante eingeführt, die sowohl den Export zur Komplilierzeit ermöglicht als auch zur Laufzeit weiter gehende Zugriffe mit Reflection. Interessanterweise erlaubt das Öffnen keine Zugriffe zur Komplilierzeit, sondern wirklich nur zur Laufzeit.

Fazit

- eine paar Dinge aus Project COIN (Sprachsyntax)
- diverse praktische Erweiterungen in den APIs
- Reactive Streams
- HTTP 2
- Modularisierung mit Project JIGSAW -- aber leider (noch) kein wirklich richtig gutes Tooling



- **Mehrmalige Verschiebung bei JDK 9**
Sept. 2016 => März 2017 => Juli 2017 => Sept. 2017
- **Lizenierung in Java 11**
- **Mal wieder weniger als geplant**
 - keine Versionierung bei JIGSAW
 - kein JSON-Support
 - statt Collection-Literals nur Convenience-Methods



Questions?

Thank You