Mehr als die Summe der Teile

Modulare Softwaresysteme mit Jigsaw

Johannes Weigend

Mit Version 9 bekommt Java endlich die lang erwartete Unterstützung für den Bau von Softwaremodulen. Das Modulsystem Jigsaw wird Teil des JDKs und der JRE-Laufzeitumgebung. Der folgende Artikel beschreibt, wie man statisch und dynamisch austauschbare Software auf der Basis von Jigsaw aufbaut, um modulare und komponentenorientierte Anwendungen zu bauen.

Java selbst nutzt das Jigsaw Platform Module System [JSR376] zur internen Modularisierung der bisher monolithischen Laufzeitumgebung (`rt.jar`). Anwendungen können Jigsaw zur Absicherung der Architekturtreue einsetzen, außerdem lassen sich Anwendungen mit minimaler JRE-Laufzeitumgebung ausliefern, welche nur die von der Anwendung benötigten Module des JDKs enthält. Mit Jigsaw lassen sich, ähnlich zu OSGi, auch Plug-in-Module schreiben, welche Anwendungen mit neuen Funktionen versehen, die zur Compilezeit noch nicht vorhanden sind.

Module

Module sind unabhängig verteilbare Einheiten (Deployment-Units), welche die Implementierung vor dem Nutzer verbergen. Der Kern der Modularisierung basiert auf dem Geheimnisprinzip. Ein Nutzer muss nicht die Details einer Implementierung kennen, um das Modul aufzurufen. Die Implementierungsdetails sind hinter einer Schnittstelle verborgen. Damit kann die Komplexität für den Nutzer auf die Komplexität der Schnittstelle reduziert werden. Alles, was der Anwender von einem Modul kennen muss, sind die öffentlichen Klassen, Interfaces und Methoden des Moduls. Die Details der Implementierung sind versteckt.

Module übertragen das public/private-Prinzip der Objektorientierung auf ganze Bibliotheken. Dieses Prinzip einer nach außen unsichtbaren Implementierung ist lang bekannt. David Parnas beschrieb das Sichtbarkeitsprinzip auf Modulebene und die Vorteile bereits 1972 [Par72].

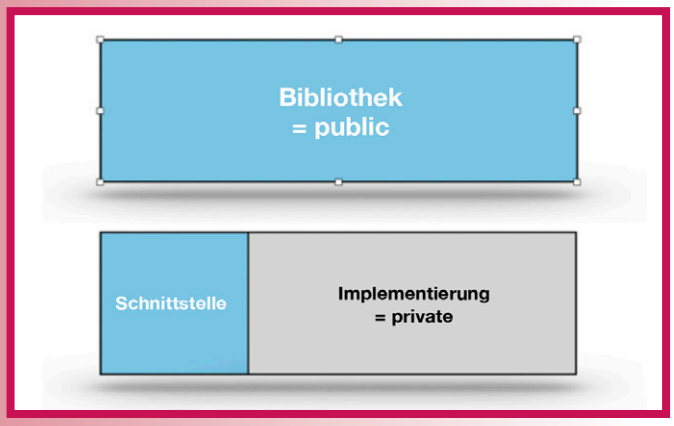

Abb. 1: Bibliothek versus Modul

Ein Modul besteht aus einem Schnittstellen- und einem Implementierungsteil in einer einzigen Deployment-Unit/Bibliothek (s. Abb. 1). Der Nutzen derartiger Kapselung ist analog zur Objektorientierung:

- Die Implementierung eines Moduls kann ohne Auswirkungen auf Nutzer geändert werden.
- Komplexe Funktionalität wird hinter einer einfachen Schnittstelle versteckt. Eine gute Testbarkeit, Wartbarkeit und Verständlichkeit ist dadurch gegeben.

Gerade heute im Zeitalter von Cloud und Microservices wird ein modulares Design zwingend notwendig. Verpackt man die zur Remote-Kommunikation notwendigen Teile der Microservices in getrennte Module und definiert die Modulschnittstellen rein fachlich, dann ist es möglich, eine Anwendung lokal oder auch verteilt auszuführen.

Will man zur Laufzeit Modulimplementierungen austauschen oder zwischen mehreren Implementierungen auswählen (Plug-in), ist es notwendig, den Schnittstellen- und den Implementierungsteil in zwei unabhängige Module aufzuspalten. Damit gibt es ein API-Modul und ein potenziell austauschbares Implementierungsmodul.

Module, die auch zur Laufzeit ausgetauscht werden können, bezeichnet man als Plug-in-Module. Das wiederum erfordert zwingend eine Trennung von Schnittstelle und Implementierung auf verschiedene Deployment-Units.

Der Bau modularer Anwendungen hat mit Java eine lange Tradition. Es gibt viele konkurrierende Ansätze zum Bau von Softwaremodulen. Allen ist aber gemeinsam, dass ein Modul als Bibliothek abgebildet wird. Bibliotheken in Java realisiert man als Sammlung von Klassen, Interfaces und weiteren Ressourcen. JARs sind ZIP-Dateien, deren Inhalt keinerlei Zugriffsrechten unterliegt. Daher bilden bisher viele Anwendungen ihre Komponenten durch einen Mix aus verschiedenen Ansätzen ab:

- Abbildung auf Paketstrukturen per Konvention,
- Abbildung auf Bibliotheken (JARs),
- Abbildung auf Bibliotheken inklusive META-Information zur Kontrolle von Abhängigkeiten und Sichtbarkeit (z. B. OSGi),
- Kontrolle von Abhängigkeiten durch Analysewerkzeuge (z. B. Sonarqube oder Structure101),
- Kontrolle von Abhängigkeiten durch Build-Werkzeuge (z. B. Maven oder Gradle) sowie
- ClassLoader-Hierarchien zur Steuerung der Sichtbarkeit zur Laufzeit (z. B. Java EE).

Alle diese Ansätze haben ihre eigenen Vor- und Nachteile. Keiner löst aber das Problem im Kern: Java kennt nativ keinen Modulbegriff. Mit Java 9 ändert sich das: Mit Jigsaw lassen sich Module bauen, die auf Ebene von JARs Sichtbarkeit und Abhängigkeiten steuern. Module stellen einen Teil ihrer Typen als Schnittstelle nach außen zur Verfügung. Die Schnittstelle eines Jigsaw-Moduls besteht aus einem oder mehreren Paketen. Per Compiler und JVM wird sichergestellt, dass kein Zugriff an der Schnittstelle vorbei auf die

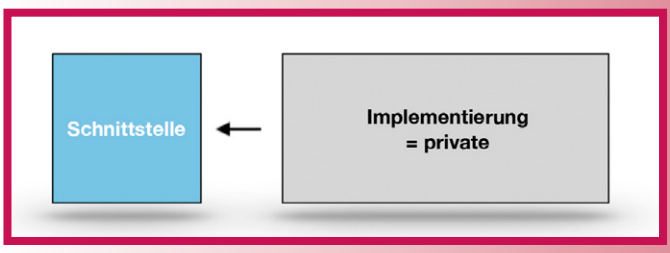

Abb. 2: Austauschbarkeit durch Trennung von Schnittstelle und Implementierung

privaten Typen (Klassen, Interfaces, Enums, Annotationen) erfolgt.

Jigsaw stellt die notwendigen Werkzeuge zur Analyse und Kontrolle von Abhängigkeiten bereit. Mit dem Analysewerkzeug `jdeps` lassen sich die Abhängigkeiten von JARs und Modulen analysieren und grafisch darstellen (per DOT/GraphViz).

Die Java 9-Laufzeitbibliotheken basieren ihrerseits auf Jigsaw. Die bisher monolithische Laufzeitbibliothek `rt.jar` ist in Java 9 in Module aufgespalten. Zyklische Abhängigkeiten in den Modulen wurden entfernt. Jigsaw erlaubt dies nicht, da zyklische Abhängigkeiten die Austauschbarkeit auf Modulebene verhindern würden.

Mit dem Werkzeug `jlink` lassen sich Anwendungen mit minimaler Java Runtime zusammenbauen. Diese Anwendungen enthalten ausschließlich die verwendeten Module aus der JDK-Modulmenge.

Der Kern von Jigsaw ist der Moduldeskriptor `module-info.java`, der vom Java-Compiler in eine Klasse `module-info.class` kompiliert wird und sich auf oberster Paketebene in jedem Jigsaw-JAR-Archiv befindet.

Der Moduldeskriptor sieht wie folgt aus [JigLang]:

```
module M @ 1.0 {
  requires A @ /* Use v2 or above */ >= 2.0 ;
  requires B for compilation, reflection;

  requires service S1;
  requires optional service S2;

  provides MI @ 4.0;
  provides service MS with C;
  exports  ME;
  permits  MF;
  class    MMain;

  view N {
    provides NI @ 1.0;
    provides service NS with D;
    exports  NE;
    permits  MF;
    class    NMain;
  }
}
```

Der Inhalt der Datei ist ein Modul mit einem Namen und einer optionalen Versionsnummer. Mittels `requires` gibt ein Modul Abhängigkeiten zu anderen Modulen an. Mittels `provides` gibt ein Modul an, dass es die Schnittstelle des angegebenen Moduls implementiert. Mit `exports` wird die Schnittstelle in Form eines Paketnamens angegeben. `permits` macht ein Modul nur für die angegebenen Module sichtbar.

Mit dem Abschnitt `view` können mehrere Sichten auf ein Modul deklariert werden. Dieser Mechanismus ist für die Unterstützung von Abwärtskompatibilität notwendig. Ein Modul kann damit mehrere Versionen eines Schnittstellenmoduls unterstützen und trotz Weiterentwicklung zu alten Modulen kompatibel bleiben.

Mail senden mit Jigsaw

Die im Folgenden entwickelte Anwendung dient zum Senden von E-Mails und besteht aus zwei Modulen:

- Das Modul `Mail` besteht aus einer öffentlichen Schnittstelle und einer privaten Implementierung. Die Schnittstelle des Moduls besteht aus einem Java-Interface sowie den Typen der Parameter und Ausnahmen. Außerdem enthält die Schnittstelle eine Fabrik (Factory Pattern) zur Erzeugung der Implementierung.
- Das Modul `MailClient` verwendet das Modul `Mail`. Es darf ausschließlich die Schnittstelle benutzen. Ein direkter Zugriff auf die Implementierungsklassen ist verboten.

Java 9 Jigsaw stellt nun sicher, dass:

- das Modul `MailClient` nur auf exportierte Klassen/Pakete im Modul `Mail` zugreift. Ein direkter Zugriff führt mit Jigsaw zu Compiler- sowie zu Laufzeitfehlern bei der Benutzung mittels Reflection-API.
- das Modul `Mail` nur die im Modul angegebenen Abhängigkeiten zu anderen Modulen nutzt.

Das entkoppelt die Modulimplementierung vom Client und macht diese änderbar. Neben der Unterstützung von Innen- und Außensicht in einem Modul stellt Jigsaw außerdem noch sicher, dass

- keine zyklischen Abhängigkeiten zwischen den Modulen entstehen. Eine Abhängigkeit vom Modul `Mail` auf das Modul `MailClient` ist damit verboten und wird durch den Compiler und die JVM überprüft.
- keine transitiven Abhängigkeiten von der Komponente `Mail` unkontrolliert an die Komponente `MailClient` weitergegeben werden. Ob die Schnittstelle abhängiger Module für den Schnittstellennutzer sichtbar ist, kann man steuern.

Abb. 3: Das Modul Mail

Das Beispiel im Quellcode

Jigsaw führt eine neue Verzeichnisstruktur für Module im Quellcode ein. Auf der obersten Ebene befinden sich nun im Source-Pfad die Module mit ihren Quellen. Das Verzeichnis entspricht dem Modulnamen. Damit kann der Java-Compiler auch abhängige Module im Quellcode finden ohne aufwendige Pfadangaben pro Modul:

```
src
   |— Mail
   |    |— de
   |    |   |— qaware
   |    |       |— mail
   |    |           |— MailSender.java
   |    |           |— MailSenderFactory.java
   |    |           |— impl
   |    |               |— MailSenderImpl.java
   |    |— module-info.java
   |— MailClient
        |— de
        |   |— qaware
        |   |— mail
        |       |— client
        |           |— MailClient.java
        |— module-info.java
```

Natürlich kann man mit Jigsaw Module beliebig ablegen. Das gezeigte Layout hat aber den Vorteil, dass alle Module mit einem einzigen Compilerlauf übersetzt werden können und nur ein Suchpfad angegeben werden muss.

Module in Java 9 Jigsaw enthalten eine spezielle Datei, den Moduldeskriptor, mit dem Namen „module-info.java“ im Default-Paket der Bibliothek. In unserem Beispiel exportiert die Mail-Komponente nur ein Paket. Die zugehörige Datei „module-info.java“ sieht wie folgt aus:

```
module Mail {
  exports de.qaware.mail;
}
```

Die Anweisung **exports** referenziert ein Paket. Über mehrere Exports-Anweisungen lassen sich mehrere Pakete als Teil der Schnittstelle definieren. In unserem Beispiel sind alle Typen im Paket „de.qaware.mail“ für einen Nutzer sichtbar. Unterpakete sind nicht sichtbar. Die Exports-Anweisung ist nicht rekursiv. Typen im Unterpaket „de.qaware.mail.impl“ sind von einem anderen Modul aus nicht zugreifbar.

Ein Nutzer des Moduls **Mail** ist das Modul **MailClient**. Der Moduldeskriptor sieht wie folgt aus:

```
module MailClient {
  requires Mail;
}
```

Die Anweisung **requires** nimmt einen Modulnamen und unterstützt optional die Angabe, ob das Mail-Modul zur Laufzeit (**requires … for reflection**) oder nur zur Compilezeit (**requires … for compilation**) sichtbar ist. Per Default bezieht sich die Requires-Anweisung sowohl auf den Java-Compiler als auch auf die JVM zur Laufzeit.

Wie im Folgenden gezeigt, nutzt der Quellcode der Komponente **MailClient** die Schnittstelle der Komponente **Mail**. Zu der Schnittstelle gehört das Java-Interface **MailSender** sowie eine Fabrik, mit der eine Implementierung erzeugt werden kann. Die Parameter für die Mail-Adresse und die Nachricht sind in diesem Beispiel einfache Java-Strings.

Jedes Jigsaw-Modul hängt automatisch von dem Java-Basismodul **java.base** ab. In diesem Modul befinden sich Basispakete wie **java.lang** oder **java.io**. Daher muss die Verwendung der Klasse **String** in diesem Beispiel nicht explizit im Modul deklariert werden:

```
package de.qaware.mail.client;

import de.qaware.mail.MailSender;
import de.qaware.mail.MailSenderFactory;

public class MailClient {
  public static void main(String [] args) {
    MailSender mail = new MailSenderFactory().create();
    mail.sendMail(
      "johannes@xzy.de", "Hello Jigsaw"
    );
  }
}
```

Wir erinnern uns: Ein Zugriff auf die privaten Implementierungsklassen ist nicht möglich. Der Versuch, die Implementierung der Klasse **MailSenderImpl** ohne Aufruf der Factory direkt mit **new** oder auch per Reflection zu erzeugen, scheitert mit der folgenden Fehlermeldung:

```
../MailClient.java:9: error: MailSenderImpl is not visible because
  package de.qaware.mail.impl is not visible
1 error
```

Das ist genau das, was wir wollen. Keiner kann nun von außen auf eine Klasse im Modul **MailSender** zugreifen mit Ausnahme der exportierten Artefakte im Paket „de.qaware.mail“. Nichtexportiert Pakete sind unsichtbar.

Damit modulare Java-Programme auch ohne externes Build-Werkzeug wie Ant, Maven oder Gradle kompiliert werden können, ist es notwendig, dass der Java-Compiler **javac** abhängige Module finden kann, auch wenn diese nur im Quellcode vorliegen. Aus diesem Grund wurde der Java-Compiler um die Angabe des Modulquellpfads erweitert. Mit der neuen Option **-modulesourcepath** wird dem Java-Compiler der Suchpfad für abhängige Module mitgeteilt. Für erfahrene Java-Programmierer ist es sehr ungewohnt, dass im Verzeichnis „src“ mehrere Module in Unterverzeichnissen liegen, die nach dem Modulnamen benannt sind. Folgt man den Konventionen im JDK, dann sind das Verzeichnisse, deren Name ein Paketname ist (z. B. „de.qaware.mail“). Das kann sehr unübersichtlich werden, hat aber den Vorteil, dass die Modulnamen global eindeutig sind. Für nicht-öffentliche Projekte spielt das aber keine Rolle. Daher verwenden wir hier fachlich sprechende Namen wie **Mail**, **MailClient** oder **MailAPI**.

Der große Vorteil an dieser neuen Codestruktur aber ist es, dass ein einziges Kommando alle Module kompilieren kann.

Vom Mail-Modul zum Mail-Plug-in

In dem obigen Beispiel ist die Schnittstelle des Moduls **Mail** eng mit der Implementierung gekoppelt. Jigsaw kennt innerhalb eines Moduls keine Sichtbarkeitsregeln zwischen Schnittstelle und Implementierung. Bidirektionale Abhängigkeiten sind hier erlaubt. Damit aus dem Modul **Mail** eine zur Laufzeit austauschbare Komponente entsteht, ist es notwendig, Schnittstelle und Implementierung in separate Module zu trennen (s. Abb. 4). Dieses klassische Plug-in-Design wird immer dann notwendig, wenn es mehrere alternative Module für ein API gibt:

```
// src/MailClient/modul-info.java
module MailClient {
  requires MailAPI;
}
// src/MailAPI/modul-info.java
module MailAPI {
  exports de.qaware.mail;
}
// src/WebMail/modul-info.java
module WebMail {
  requires MailAPI;
}
```

Das Modul **MailClient** hängt jetzt von dem neuen Modul **MailAPI** ab. Das Modul **MailAPI** exportiert die Schnittstelle, hat aber selbst keine Implementierung. Implementiert wird diese Schnittstelle von einem dritten Modul, dem Modul **WebMail**, das nichts exportiert, sondern die Schnittstelle implementiert. Zur Compilezeit wäre es ausreichend, dass sowohl Client als auch Implementierungsmodul das API-Modul per **requires** deklarieren.

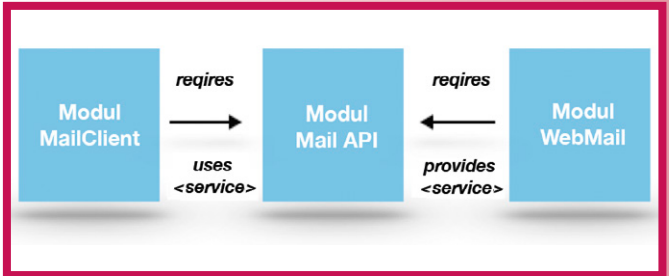

Abb. 4: Das Mail-Modul als austauschbares Plug-in

Zur Laufzeit hat man aber damit ein Problem, da die Implementierungsklassen im Modul **WebMail** versteckt sind. Ein weiteres Problem ist nun, dass sich die Fabrik im Modul **MailAPI** befindet muss, damit der Client diese nutzen kann. Da die Factory aber eine Abhängigkeit zur Implementierung benötigt, bekommt man eine zyklische Abhängigkeit, was zu einem Übersetzungsfehler führt. Die Frage ist: „Wie kann man eine versteckte Implementierungsklasse erzeugen?“

Zur Erzeugung von Modul-privaten Implementierungsklassen wurde im JDK 9 die Klasse **ServiceLoader** aus dem Paket java.util erweitert. Über die Angabe **provides** in dem Moduldeskriptor des Implementierungsmoduls kann man ein Service-Interface mit einer privaten Implementierungsklasse verbinden. Der **ServiceLoader** kann damit auf die Implementierungsklasse zugreifen und diese instanziieren. Eine Erzeugung per Reflection mit **Class.forName().newInstance()** ist nicht möglich. Diese Entscheidung hat Auswirkungen auf alle Dependency-Injection-Frameworks wie Spring oder Guice. Die heutigen Implementierungen dieser Frameworks müssen für Jigsaw auf den ServiceLoader-Mechanismus angepasst werden.

Das nutzende Modul deklariert die Nutzung eines Service über die Angabe von **uses**. Das implementierende Modul gibt mittels **provides with** an, welche Implementierung durch den ServiceLoader erzeugt werden darf. Damit funktioniert die Instanziierung in einem nutzenden Modul per ServiceLoader:

```
// src/MailAPI/modul-info.java
module MailAPI {
  exports de.qaware.mail;
}
// src/MailClient/modul-info.java
module MailClient {
  requires MailAPI;
  uses de.qaware.mail.MailSender
}

// src/WebMail/modul-info.java
module WebMail {
  requires MailAPI;
  provides de.qaware.mail.MailSender with
           de.qaware.mail.smtp.SmtpSenderImpl;
}
// src/MailClient/de/qaware/mail/client/MailClient.java

// OK: Create implementation by using the java.util.ServiceLoader
MailSender mail = ServiceLoader.load(MailSender.class)
  .iterator().next();

// NOK: Reflection is not allowed:
// mail = (MailSender) Class.forName(
                "de.qaware.mail.impl.MailSenderImpl")
                       .getConstructors()[0]
                       .newInstance();

// NOK: Direct instantiation is not allowed:
// mail = new de.qaware.mail.impl.MailSenderImpl();
```

Eine Deklaration des Service im META-INF-Verzeichnis ist hier nicht mehr notwendig. Die direkte Nutzung per Reflection ist aber nach wie vor verboten und wird durch einen Laufzeitfehler unterbunden. Ebenso ist die Implementierungsklasse natürlich nach wie vor privat und kann nicht direkt genutzt werden.

Der Modulpfad und automatische Module

Java 9 unterstützt die Angabe von Modulen bei der Ausführung. Aus Gründen der Abwärtskompatibilität wurde ein neuer Lademechanismus für Modul-JARs eingeführt: der Modulpfad (modulepath). Genau wie beim Klassenpfad (classpath) können JARs und/oder ganze Verzeichnisse angegeben werden, aus denen Module geladen werden. Befinden sich JARs im Modulpfad ohne Moduldeskriptor, wird zur Laufzeit ein Moduldeskriptor automatisch generiert, der alles exportiert und das Modul als Abhängigkeit allen anderen Modulen hinzufügt. Ein solches Modul wird als „automatic module“ bezeichnet. Damit wird eine Koexistenz von Jigsaw-Modulen mit normalen JARs gewährleistet. Beide können sogar im gleichen Verzeichnis abgelegt werden:

```
# run
java -mp mlib -m MailClient
```

Module bauen, paketieren und ausführen

Mit einem einzigen Kommando lassen sich alle Module einer Anwendung kompilieren und strukturiert in einem Ausgabeverzeichnis ablegen:

```
# compile
javac -d build -modulesourcepath src $(find src -name "*.java")
```

Das Kommando übersetzt die Module unter der Pfadwurzel src und speichert die generierten Klassen in der identischen Verzeichnisstruktur in den Pfad ./build. Der Inhalt des .build-Verzeichnisses kann nun in getrennte JAR-Dateien gepackt werden. Die Angabe der Startklasse (**--main-class**) ist optional:

```
# pack
jar --create
  --file mlib/WebMail@1.0.jar
  --module-version 1.0
  -C build/WebMail .

jar --create
  --file mlib/MailAPI@1.0.jar
  --module-version 1.0
  -C build/MailAPI .

jar --create
  --file mlib/MailClient@1.0.jar
  --module-version 1.0
  --main-class de.qaware.mail.client.MailClient
  -C build/MailClient .
```

Im Ausgabeverzeichnis mlib liegen nun drei Module. Durch die Angabe dieses Pfads als Modulpfad ist die JVM in der Lage, die Anwendung zu starten:

```
# run
java -mp mlib -m MailClient
```

Modulare Anwendungen ausliefern

Um eine startbare Anwendung auszuliefern, musste man bisher eine Java Runtime (JRE) mitliefern. Für die Anwendung selbst waren bisher Startskripte erforderlich, die den Klassenpfad definiert haben, um die Anwendung mit ihren abhängigen Bibliotheken korrekt starten zu können. Das JRE hat immer die gesamte Java-Funktionalität mitgeliefert, auch wenn nur ein kleiner Teil von der Anwendung verwendet wurde.

Mit Java 9 gibt es nun das Kommando **jlink**. Damit lassen sich Anwendungen mit den benötigten Teilen des JDKs zusammenbauen. Nur die wirklich benötigten Module werden verwendet, sodass eine minimale Java-Laufzeitumgebung bereitgestellt

wird. Verwendet eine Anwendung beispielsweise kein CORBA, wird dieses Modul auch nicht kopiert:

```
# link
jlink --modulepath $JAVA_HOME/jmods:mlib --addmods MailClient,Mail
  --output mailclient
```

Die Anwendung kann nun mit einem einzigen Skript gestartet werden. Ein Wissen über Module und deren Abhängigkeiten ist nicht notwendig. Das durch `jlink` generierte Ausgabeverzeichnis sieht wie folgt aus:

```
.
|— bin
|   |— MailClient
|   |— java
|   |— keytool
|— conf
|   |— net.properties
|   |— security
|       |— java.policy
|       |— java.security
|— lib
...
```

Der Verzeichnisbaum zeigt die komplette minimale Laufzeitumgebung der Anwendung. Im bin-Verzeichnis befindet sich das generierte Startskript, mit dem die Anwendung ohne Angabe von Parametern gestartet werden kann. Alle verwendeten Module werden automatisch in einer einzigen Datei zusammengepackt. Durch den Aufruf des `MailClient`-Startskripts im bin-Verzeichnis kann nun die Anwendung gestartet werden:

```
cd mailclient/bin
./MailClient
Sending mail to: x@x.de message: A message from JavaModule System
```

Zusammenfassung

Mit Jigsaw hat das Team um Marc Reinhold bei Oracle einen großen Wurf geleistet. Mit Jigsaw können modulare Softwaresysteme rein auf Basis von Java-Bordmitteln entwickelt werden. Die Auswirkungen auf bestehende Tools und Entwicklungsumgebungen sind groß. Es wird daher noch einige Zeit dauern, bis die gängigen Entwicklungsumgebungen und Build-Systeme Jigsaw umfassend unterstützen werden. Da Jigsaw aber nun Teil von Java 9 ist, kann man sicher sein, dass dies passiert. Unreife Werkzeugunterstützung, wie das bei OSGi der Fall war, gehört damit wohl der Vergangenheit an.

Da Jigsaw uns nicht den Job abnimmt, Module sinnvoll zu entwerfen, zu entwickeln und zu testen, werden wir auch in Zukunft trotz Jigsaw von monolithischer, schwer wartbarer Software umgeben sein. Wer aber die neuen Mechanismen diszipliniert nutzt, wird flexiblere und besser wartbare Software bauen können.

Literatur und Links

[JigLang] `http://openjdk.java.net/projects/jigsaw/doc/lang-vm.html`

[JSR376] Java Specification Request 376: Java Platform Module System,
`http://openjdk.java.net/projects/jigsaw/spec/`

[Par72] D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, in: CACM, Dezember, 1972,
`https://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf`

Johannes Weigend ist technischer Geschäftsführer und Architekt bei QAware in München. Er hält Vorlesungen an der Hochschule Rosenheim und verantwortet den Bereich F&E bei QAware. Johannes Weigend wurde als Java Rockstar 2015 nominiert und ist Mitglied im NetBeans Dream Team. E-Mail: johannes.weigend@qaware.de