Transfer der resultierenden `de.firma.moda-1.0.jar` zum Computer von Team 2 in das modules-Verzeichnis und dann auf dem zweiten Computer:

```
1. javac -p modules -d classes\de.firma.modmain
     ↪ src\de.firma.modmain\*.java
     ↪ src\de.firma.modmain\de\firma\modmain\*.java
2. jar --create
     ↪ --file modules/de.firma.modmain-1.0.jar
     ↪ --module-version 1.0
     ↪ --main-class de.firma.modmain.modmain
     ↪ -C classes/de.firma.modmain .
```

Ausführung der Anwendung:

```
java -p modules -m de.firma.modmain
```

Was in der praktischen Arbeit mit Modulen immer wieder vorkommt, ist der Umstand, dass verschiedene Module, die gleichen Abhängigkeiten benötigen. Bei der Modellierung solcher Beziehungen, hilft die Unterstützung von transitiven Abhängigkeiten durch das Modulsystem, wie sie im folgenden Kapitel vorgestellt wird.

3.2.2 Transitive Abhängigkeiten

Die im vorangegangenen Beispiel verwendeten Module `de.firma.modmain` und `de.firma.moda` werden um die beiden weiteren Module `de.firma.modb` und `de.firma.modc` ergänzt, wobei Modul `de.firma.moda` von diesen beiden abhängt. Modul `de.firma.modmain` soll in diesem Beispiel nun ebenfalls Klassen aus `de.firma.modc` verwenden können. Dies lässt sich durch die Deklaration einer transitiven Abhängigkeit modellieren. Abbildung 3–11 zeigt den Modulgraphen.

Zunächst wird eine passende Verzeichnisstrukur erzeugt:

```
1. cd \MeinWorkspace
2. mkdir FourModulesTransDep
3. cd FourModulesTransDep
4. mkdir src\de.firma.modmain\de\firma\modmain
5. mkdir src\de.firma.moda\de\firma\moda
6. mkdir src\de.firma.modb\de\firma\modb
7. mkdir src\de.firma.modc\de\firma\modc
```

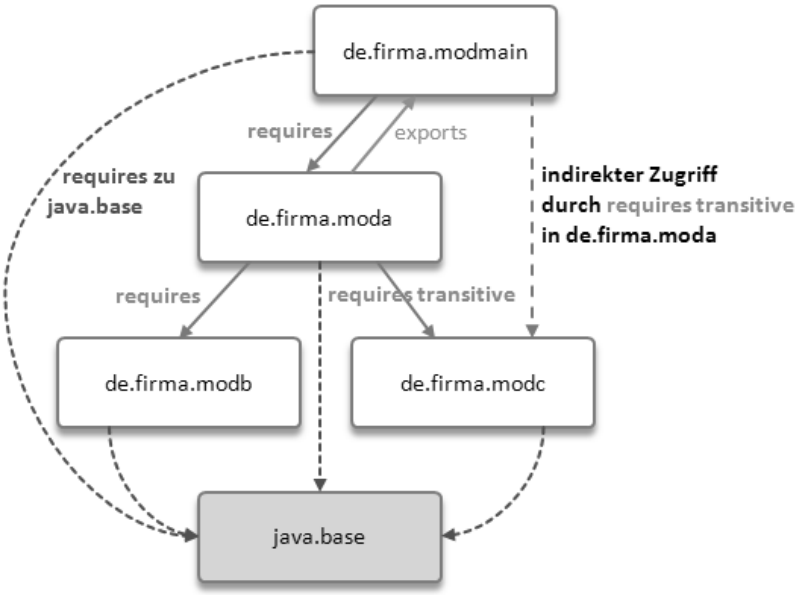

Abb. 3–11
Deklaration und Implementierung eines Java Moduls

Im Moduldeskriptor von Modul `de.firma.modmain` hat sich nichts geändert, wie folgendes Listing zeigt:

Listing 3–6
module-info.java

```
module de.firma.modmain {
   // benötigtes Modul
   requires de.firma.moda;
}
```

Der Moduldeskriptor von Modul `de.firma.moda` wurde um die beiden benötigten Abhängigkeiten erweitert:

Listing 3–7
module-info.java

```
module de.firma.moda {
   // benötigtes Module
   requires de.firma.modb;
   requires transitive de.firma.modc;

   // exportiertes Paket
   exports de.firma.moda;
}
```

Zudem zeigt das Listing eine mit **requires transitive** deklarierte transitive Abhängigkeit. Dadurch wird das Modul `de.firma.modc` für jedes andere Modul zugreifbar, welches Modul `de.firma.moda` importiert.

Transitive Abhängigkeiten helfen bei einer sauberen Strukturierung, da alle neben `moda` für `modmain` noch benötigten Abhängigkeiten vollständig mit der Deklaration von `moda` gekapselt werden können. Des

Weiteren hilft dieser Mechanismus bei der Erzeugung von Aggregator-Modulen; also Module, die einzig und alleine dafür da sind, eine Menge bestimmter Module zusammenzuführen und gebündelt nach außen anzubieten. Dadurch müssen anderen Module nur dieses eine Aggregatormodul lesen, ohne eine ganze Phalanx an Abhängigkeiten deklarieren zu müssen. Als Beispiele wären hier bei einer Schichtenarchitektur die mögliche Zusammenfassung einer Schicht als Modul zu nennen oder die mit Java 8 eingeführten Compact-Profiles, die im Kapitel über das modularisierte JDK nochmals genauer betrachtet werden.

Die entsprechenden Moduldeskriptoren der Module `de.firma.modb` und `de.firma.modc` sehen dabei wie folgt aus:

Listing 3–8 *module-info.java Dateien*

```
module de.firma.modb {
   // exportiertes Paket
   exports de.firma.modb;
}

module de.firma.modc {
   // exportiertes Paket
   exports de.firma.modc;
}
```

Im vorliegenden Beispiel bekommen die Module `de.firma.modb` und `de.firma.modc` noch jeweils eine Klasse. Modul `de.firma.modb` enthält die Klasse `TestB`:

Listing 3–9 *TestB.java*

```
package de.firma.modb;

public class TestB {
         public static String getName() {
      return TestB.class.getName();
   }
}
```

Und Modul `de.firma.modc` enthält die Klasse `TestC`:

Listing 3–10 *TestC.java*

```
package de.firma.modc;

public class TestC {
   public static String getName() {
      return TestC.class.getName();
   }
}
```

Die Main-Methode der Startklasse `App` wird nun um einen Zugriff auf das Modul `de.firma.modc` erweitert, obwohl in dem Moduldeskriptor von `de.firma.modmain` keine entsprechende Abhängigkeit explizit deklariert wurde:

Listing 3–11
App.java

```
package de.firma.modmain;

import de.firma.moda.TestA;
import de.firma.modc.TestC;

public class App {
  public static void main(String[] args) {
    System.out.println("Name des importierten
                Moduls A: " + TestA.getName());
                System.out.println("Name des transitiv
                erreichbaren Moduls C: " + TestC.getName());
  }
}
```

Transitive Abhängigkeiten

Transitive Abhängigkeiten

Transitive Abhängigkeiten werden im Moduldeskriptor `module-info.java` mit **requires transitive** `de.firma.meinModul` deklariert. Wenn die Abhängigkeit nur für bestimmte Module weitergereicht werden soll, kann auch **requires transitive to** geschrieben werden, mit dem entsprechenden Modul hinter **to**.

Beispiel:

```
module de.firma.moda {
  requires transitive de.firma.modb;
  requires transitive de.firma.modc to de.firma.modmain;
}
```

Neben den fest deklarierten Abhängigkeiten zwischen Modulen, die bereits statisch während der Kompilierung vorliegen, gibt es auch einen Mechanismus, wo die direkte Abhängigkeit erst zur Laufzeit aufgelöst wird und was Thema des nächsten Kapitels ist.

3.3 Services

Seit Java SE 6 existiert in der Spezifikation ein Service-Provider-Mechanismus, der eine bessere Entkopplung zwischen Klassen unterstützt, indem Klassenabhängigkeiten erst zur Laufzeit aufgelöst werden. Dadurch wird eine erheblich stärkere Entkopplung zwischen den Provider-Klassen und den später auf diese zugreifenden Klassen erreicht. Letzte-

Modulübergreifender Ressourcenzugriff

Das Paket, in welchem die Ressource liegt, die für andere Module zugreifbar sein soll, mit **opens** nach außen öffnen und wie folgt zugreifen:

```
InputStream inputStream = module.getResourceAsStream(
  "[Paket ohne führendem Schrägstrich]/[Dateiname]")
```

Nachdem nun der Ressourcen-Zugriff gezeigt wurde und in den Kapiteln davor, wie Module generell erstellt und genutzt werden können, stellt sich die Frage, wie mit JAR-Bibliotheken umgegangen wird, die keine Module sind, oder wie solche Nicht-Module in Modulen verwendet werden können. Diese Möglichkeit ist wichtig für die Abwärtskompatibilität von Java und für die Nutzung von Modulen in Anwendungen, die mit Java vor Version 9 erstellt wurden. Das nächste Kapitel zeigt, wie diese Fragen gelöst wurden.

3.5 Arten von Modulen

Das Java-Modulsystem unterscheidet folgende fünf Arten von Modulen:

- Platform Explicit Modules
- Application Explicit Modules
- Automatic Modules
- Open Modules
- Unnamed Module

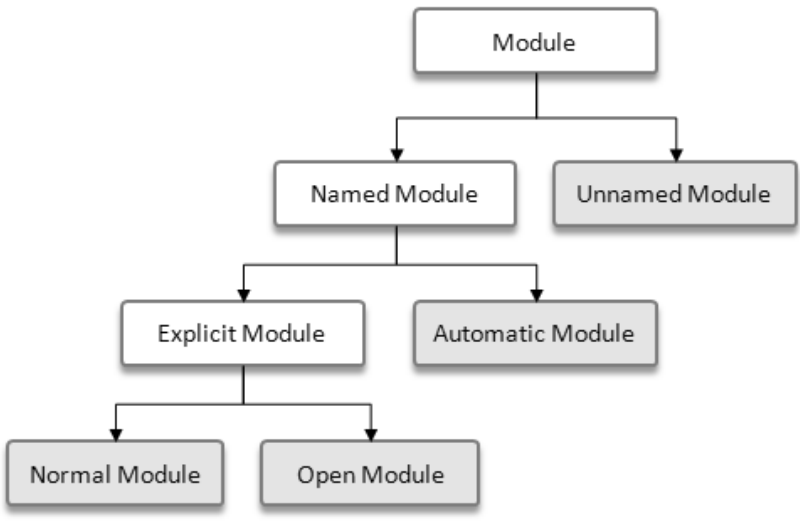

Abb. 3–19
Modularten

Alle Module außer der Modulart **Unnamed Module** haben einen Modulnamen und sind daher der Gruppe der **Named Modules** zuzuordnen. Die Gruppe **Named Modules** teilt sich in **Explicit Modules** und **Automatic Modules** auf. Bei Letzteren handelt es sich um gewöhnliche JARs, die im Modulpfad liegend vom Modulsystem automatisch als Module behandelt werden. Diese bringen keinen eigenen Moduldeskriptor mit im Gegensatz zu den **Explicit Modules**. Letztere werden nochmals in normale Module und **Open Modules** unterteilt. Die **Open Modules** sind zur Laufzeit für jeglichen Reflection-Zugriff geöffnet. Abbildung 3–19 liefert eine Überblickshierarchie über alle Modularten, die nicht zu den Java-Plattform-Modulen gehören.

Die nächsten Kapitel beleuchten kurz das Wesen der einzelnen Modularten und erläutern, wie diese miteinander im Zusammenhang stehen und zudem für eine Abwärtskompatibilität von Java sorgen und wie die Modularten die Softwaremigration unterstützen.

3.5.1 Platform Explicit Modules

Die Java Runtime selbst ist seit Java 9 ebenfalls modularisiert, wobei hier von den sogenannten Plattform-Modulen gesprochen wird. Weitere Details lassen sich dem Kapitel über das modularisierte JDK entnehmen, insbesondere wie individuell zusammengestellte Java Runtimes erstellt werden können. Das einzige Plattform-Modul, das immer benötigt wird, ist das `java.base`-Modul, welches die Kernfunktionalität von Java enthält, sowie das Java-Modulsystem selbst. Bei diesem Modul handelt es sich um ein **Aggregatormodul**, dass eine Reihe von Modulen kapselt.

```
module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
          exports sun.net.www to
                        java.desktop,
                        javafx.web,
                        jdk.deploy
    ...
}
```

Der Ausschnitt des Moduldeskriptors zeigt, dass ein Teil der Modulnamen mit dem Präfix `java` beginnen und andere mit `jdk`. Alle in der Java SE 9-Plattform-Spezifikation definierten Module, auf die innerhalb von Anwendungen direkt zugegriffen wird, beginnen mit `java`, und alle anderen, die wichtig für das JDK sind, beginnnen mit `jdk`.

3.5.2 Application Explicit Modules

Bei Modulen dieser Art handelt es sich um Module von Anwendungen, die einen Moduldeskriptor enthalten (*module-info.java*) mit allen deklarierten Abhängigkeiten und Sichtbarkeiten. Bei den in den bisherigen Beispielen erstellten Modulen, handelt es sich um eben diese, die gepackt auch als **Modular Jars** bezeichnet werden.

Es reicht also ein Moduldeskriptor, wie z. B. folgender, und die Verortung des resultierenden JARs in den Modulpfad, um ein Application Explicit Module zu erhalten.

```
module de.firma.moda {
        exports de.firma.moda;
}
```

3.5.3 Automatic Modules

Diese Module sind besonders wichtig für die Abwärtskompatibilität und für die Migration von Anwendungen hin zu einer auf Java-Modulen basierenden, modularisierten Architektur. Wenn JAR-Bibliotheken, die keinen Moduldeskriptor besitzen, in den Modulpfad gelegt werden, dann werden diese JARs als sogenannte Automatic Modules behandelt. Das bedeutet, dass diese JARs, die aufgrund des fehlenden Moduldeskriptors eigentlich keine Module darstellen, aufgrund ihrer Positionierung im Modulpfad als solche speziellen Module behandelt werden. Das Modulsystem verwendet den JAR-Dateinamen als Modulnamen (ohne Versionsnummer und Endung) und exportiert automatisch alle enthaltenen Pakete. Zusätzlich importiert das Automatic Module alle anderen verfügbaren Module und kann somit auf alle exportierten Pakete anderer Module zugreifen. Application Explicit Modules können umgekehrt auf Automatic Modules zugreifen.

Beispiel: Die bekannte Bibliothek `guava-22.0.jar` wird in den Modulpfad gelegt. Das JAR enthält keinen Moduldeskriptor und das Java-Modulsystem behandelt diese Bibliothek beim Start als Automatic Module und weist diesem den Modulnamen `guava` zu.

Der Moduldeskriptor eines Application Explicit Modules, der diese Bibliothek nutzen möchte, könnte dann so aussehen:

```
module de.firma.moda {
        exports de.firma.moda;
        requires guava;
}
```

3.5.4 Namensfestlegung für Automatic Modules

Neben der automatischen Namenszuweisung für Automatic Modules durch das Java-Modulsystem ist es möglich, den Namen festzulegen. Hierfür wird die `MANIFEST.MF`-Datei des JARs um den Eintrag `Automatic-Module-Name` ergänzt.

Eine gültige Manifest-Datei könnte dann wie folgt aussehen:

Listing 3–31
MANIFEST.MF

```
Manifest-Version: 1.0
Created-By: Apache Maven 3.5.0
Build-Jdk: 9
Automatic-Module-Name: supermod
```

Ein JAR mit diesem Manifest, welches auf dem Modulpfad liegend als Automatic Module behandelt wird, würde dann von anderen Modulen über `requires supermod` referenziert werden können.

Das folgende Beispiel zeigt das Modul `de.firma.modmain` und das als Automatic Module behandelte JAR `modauto-1.0-SNAPSHOT.jar`. Letzterem würde vom Modulsystem automatisch der Name `modauto` zugewiesen werden, aber im Beispiel wird beim Bau der Anwendung der Name `supermod` festgelegt.

Das Automatic Module besteht nur aus folgender Klasse:

```
package de.firma.modauto;

public class Test {
  public static String getInfo(){
    return "Info von de.firma.modauto";
  }
}
```

Das Explicit Module bekommt folgenden Moduldeskriptor und die ausführbare Klasse `App`:

```
module de.firma.modmain {
  requires supermod;
}
```

```
package de.firma.modmain;

import de.firma.modauto.Test;

public class App {
  public static void main(String[] args) {
    System.out.println("modmain called");
    System.out.println(Test.getInfo());
  }
}
```

Beim Verpacken der Klasse `Test` in ein JAR, soll das Manifest automatisch erweitert werden. Hier für wird die Datei `manifestModAuto.txt` mit dem entsprechenden Eintrag erzeugt:

Listing 3–32 *manifestModAuto.txt*

```
Automatic-Module-Name: supermod
```

Zu beachten ist, dass hinter der namensgebenden Zeile zwingend ein Zeilenumbruch erfolgen muss, da die Zeile sonst nicht in das erzeugte Manifest aufgenommen wird.

Die Kompilierung und das Verpacken des Automatic Modules erfolgt wie folgt:

```
javac -d classes\modauto
    ↪ modauto\src\main\java\de.firma.modauto
    ↪ \de\firma\modauto\*.java
jar --create
    ↪ --file modules\modauto.jar
    ↪ --manifest=manifestModAuto.txt
    ↪ -C classes\modauto .
```

Danach wird das Explicit Module erstellt:

```
javac -p modules
    ↪ -d classes
    ↪ --module-source-path modmain\src\main\java
    ↪ modmain\src\main\java\de.firma.modmain\*.java
    ↪ modmain\src\main\java\de.firma.modmain
    ↪ \de\firma\modmain\*.java
jar --create
    ↪ --file modules\de.firma.modmain.jar
    ↪ --main-class de.firma.modmain.App
    ↪ -C classes\de.firma.modmain .
```

Gestartet werden kann die Anwendung dann auf diesem Wege:

```
java -p modules -m de.firma.modmain
```

Bei der zum Buch vorhandenen Quellcode-Sammlung liegt dieses Beispiel zusätzlich als Maven-Variante vor.

3.5.5 Open Modules

Grundsätzlich gilt bei der Entwicklung von Modulen: Was nicht exportiert wird, ist nach außen auch nicht sichtbar. Dieser restriktive Zugriffsschutz ist bei der Anwendungsentwicklung richtig und genügt dem Modularisierungsprinzip. Nun ist es in der realen Java-Welt aber so, dass sich viele Konzepte und Frameworks wie z. B. Context and Dependency Injection (CDI) und die Persistierung mittels der Java Persistence API (JPA) etabliert haben, die fleißig Gebrauch machen vom Klassenzugriff per Reflection. Dazu gesellen sich viele Implementierungen vergangener Zeit, die als Teil einer bestehenden Anwendung vielleicht hin zu Modulen migriert werden sollen. Bei der Entstehung der Spezifikation des Java-Modulsystems mit dem Projekt Jigsaw wurde dieser Punkt über einen langen Zeitraum intensiv diskutiert. Beim trivialen Ansatz wird das Paket, auf welches mit Reflection zugegriffen werden soll, einfach exportiert. Dies würde aber in den meisten Fällen der gewünschten Modulkapselung völlig zuwiderlaufen. Aus diesem Grund wurden die Open Modules ersonnen.

Diese Module sind ähnlich zu den Explicit-Modules, aber mit dem Unterschied, dass **zur Laufzeit** alle Pakete für **Deep Reflection** exportiert bzw. zugänglich gemacht werden. Mit Deep Reflection ist der Zugriff auch auf nicht öffentliche Typen gemeint, also die Möglichkeit des kompletten Zugriffs per Reflections, wie er auch vor Java 9 genutzt werden konnte. Diese Module weichen das Konzept der starken Kapselung ganz offensichtlich auf, wurden aber nach langen Diskussionen als diese zusätzliche Modulart in die Spezifikation aufgenommen, was sich darauf begründet, dass gerade bei der Migration viele Module aufgetaucht sind, die den Reflections-Mechanismus benötigen. Bei der modularen Softwareentwicklung sollte allerdings sehr genau überlegt werden, ob der Zugriff per Reflections wirklich in eigenen Modulen zulässig sein soll. Diese Art von Modulen ist neben den Automatic Modules insbesondere bei der Migration von Anwendungen hin zu einer modularisierten Form hilfreich.

Der folgende Moduldeskriptor zeigt, wie ein Modul als Open Modul deklariert wird, wodurch alle seine Pakete automatisch für den Zugriff durch Deep Reflection freigegeben sind:

```
open module de.firma.moda {
  // alle Pakete für Deep Reflection geöffnet
}
```

Eine andere Möglichkeit ist die Freigabe von Pakete für Deep Reflections innerhalb von Nicht-Open-Modules:

```
module de.firma.moda {
  // Paket sichtbar nach außen
  exports de.firma.moda.paketa;
  // opens erlaubt Zugriff per Deep Reflection
  opens de.firma.moda.internal;
  // Paket sichtbar nach außen und Zugriff
  // per Deep Reflection möglich
  exports de.firma.moda.paketb;
  opens de.firma.moda.paketb;
}
```

Das Paket `paketa` des Moduls `de.firma.moda` wird zur Kompilierungs- und Laufzeit exportiert. Dahingegen ist das Paket `internal` nach außen nicht sichtbar, aber zur Laufzeit für Deep-Reflection-Zugriffe geöffnet. Auch lassen sich beide Schlüsselwörter für das gleiche Paket verwenden. Im obigen Moduldeskriptor ist das Paket `paketb` nach außen sichtbar und wird zur Kompilierungs- und Laufzeit exportiert und ist ebenfalls für Deep Reflection geöffnet.

3.5.6 Unnamed Module

Wie in einem vorherigen Abschnitt bereits erläutert, ist es möglich, neben dem Modulpfad auch weiter den klassischen Classpath zu nutzen. Falls nun ein Classpath definiert ist und sich in diesem Klassen und JAR-Bibliotheken befinden, dann werde diese alle zusammen dem sogenannten Unnamed Module zugeordnet. JAR-Bibliotheken im Classpath können keine Java-Module sein, selbst wenn diese über einen Moduldeskriptor verfügen sollten. Echte Java-Module müssen also immer im Modulpfad liegen, selbst wenn der Classpath mit genutzt werden soll. Alle Klassen im Unnamed Module können untereinander beliebig aufeinander zugreifen, wie es auch aus Java bis Version 8 bekannt ist. Zudem kann das Unnamed Module auf alle anderen Java-Module zugreifen. Der umgekehrte Weg, also der Zugriff auf das Unnamed Module, ist nur den Automatic Modules gestattet, aber nicht den Application Explicit Modules. Dies macht auch Sinn, da Application Explicit Modules das Modul, auf welches sie zugreifen wollen, explizit lesen müssen (`requires [Modulname]`) und das Unnamed Module hat keinen

Namen. Zudem wäre eine zuverlässige Konfiguration einschließlich des Modulgraphen gar nicht möglich. Pro Classloader kann es maximal ein Unnamed Module geben. Um dem potenziellen Problem der Namensgleichheit von Paketen im Classpath und Modulpfad vorzubeugen, gibt es zudem die Regel, dass Named Modules immer dem Unnamed Module gegenüber bevorzugt werden. Wenn also ein Paket sowohl in einem Named Module, wie auch in einem Unnamed Module liegt, wird immer das Paket im Named Module gelesen. Dies ist wichtig, um weiterhin einen eindeutigen Abhängigkeitsgraphen bilden zu können.

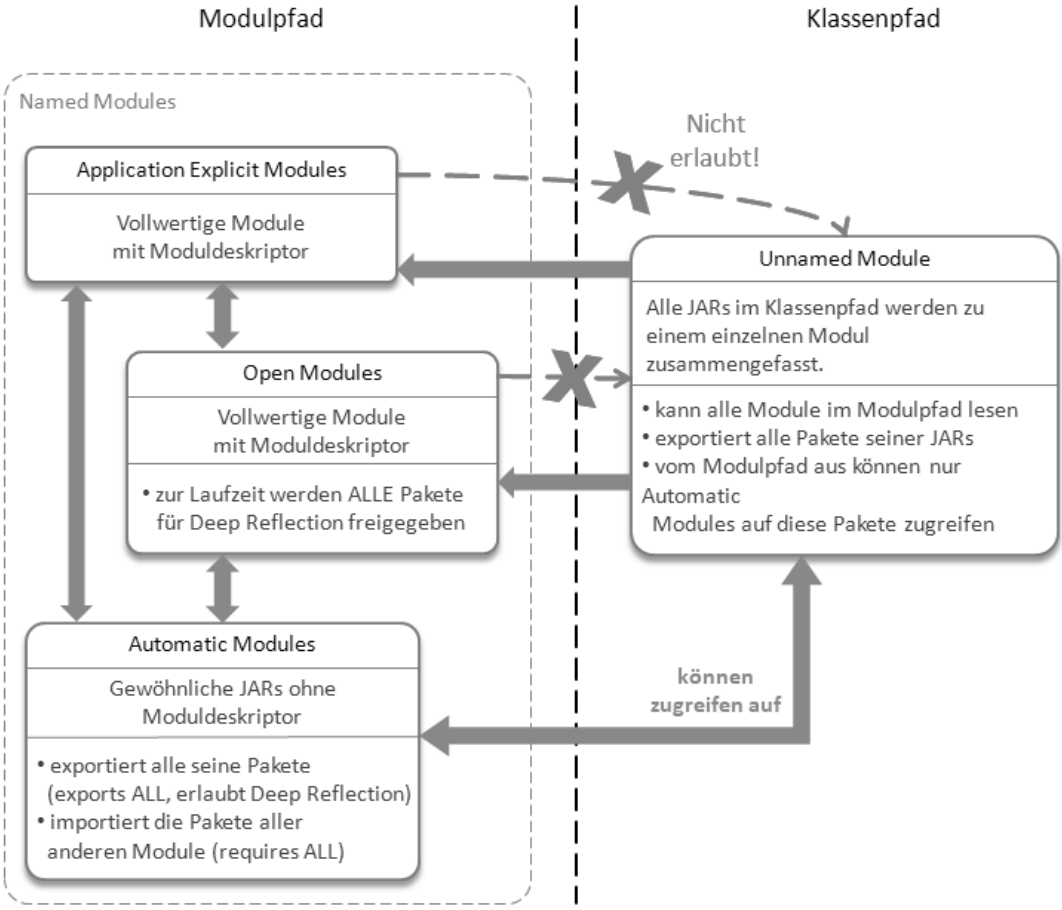

Abb. 3–20
Arten von Modulen mit Zugriffsrechten