12 Übungen zur Modularisierung

Mithilfe von Übungen sollen Module, deren Definition und Kombination, aber auch das Deployment modularer Applikationen kennengelernt werden.

✍ Aufgabe 1 – Applikation schrittweise modularisieren ✍

Gegeben sei eine einfache Applikation. Dort ist die Hauptfunktionalität in der `main()`-Methode und die eigentliche Berechnung in einer Utility-Methode momentan noch monolithisch innerhalb einer Klasse realisiert. Diese soll nun schrittweise mit dem Modularisierungsansatz aus JDK 9 anhand der folgenden Unteraufgaben in eine modularisierte Applikation umgewandelt werden.

```
package com.timeexample;

import java.time.LocalDateTime;

public class CurrentTimeExample
{
    public static void main(final String[] args)
    {
        System.out.println("Now: " + getCurrentTime());
    }

    public static LocalDateTime getCurrentTime()
    {
        return LocalDateTime.now();
    }
}
```

Aufgabe 1a: Strukturierung zur Modularisierung Extrahieren Sie aus der Klasse die Klassen `TimeInfoApplication` und `TimeUtils`. Nutzen Sie als Strukturierung die Packages `app` und `services`:

```
`-- src
    `-- com
        `-- timeexample
            |-- app
            |   `-- TimeInfoApplication.java
            `-- services
                `-- TimeUtils.java
```

Folgende Kommandos dienen zum Kompilieren und Starten:

```
javac src/com/timeexample/app/TimeInfoApplication.java \
      src/com/timeexample/services/TimeUtils.java -d build

java -cp build com.timeexample.app.TimeInfoApplication
```

Als Ausgabe sollten Sie in etwa Folgendes erhalten:

```
Now: 2017-03-10T18:53:55.005843
```

Die Applikation ist nun vorbereitet, um in zwei Komponenten unterteilt zu werden.

Aufgabe 1b: Applikation modularisieren Überführen Sie die Verzeichnisse in zwei Module `timeclient` und `timeserver`. Nutzen Sie dazu jeweils eine `module-info.java`-Datei zur Moduldefinition, in der die Abhängigkeiten korrekt spezifiziert sind. Kompilieren Sie die modularisierte Applikation in das Ausgabeverzeichnis `build`. Das Verzeichnis sollte danach in etwa wie folgt aussehen:

```
'-- build
    |-- timeclient
    |   |-- com
    |   |   '-- timeexample
    |   |       '-- app
    |   |           '-- TimeInfoApplication.class
    |   '-- module-info.class
    '-- timeserver
        |-- com
        |   '-- timeexample
        |       '-- services
        |           '-- TimeUtils.class
        '-- module-info.class
```

Starten Sie zum Prüfen der Funktionalität die modularisierte Applikation wie folgt:

```
java -p build -m timeclient/com.timeexample.app.TimeInfoApplication
```

Als Ausgabe sollten Sie in etwa Folgendes erhalten:

```
Now: 2017-03-10T19:10:24.122353
```

Tipp Starten Sie mit dem unabhängigen Kompilieren jedes der beiden Module:

```
javac -d build/timeserver \
      src/timeserver/module-info.java \
      src/timeserver/com/timeexample/services/TimeUtils.java

javac -d build/timeclient \
      src/timeclient/module-info.java \
      src/timeclient/com/timeexample/app/TimeInfoApplication.java
```

Beim Kompilieren könnte es zu Fehlern kommen:

1. Denken Sie daran, den Module-Path anzugeben, etwa `-p build/timeserver`.
2. Prüfen Sie die Moduldeskriptoren auf Abhängigkeiten und Freigaben, wie z. B. `requires timeserver` und `exports com.timeexample.services`.

Aufgabe 1c: Modulare JARs erstellen Überführen Sie die Applikation in zwei modulare JARs und legen Sie dazu ein Verzeichnis `lib` an, das als Ziel für die Module dient:

```
'-- lib
    |-- timeclient.jar
    '-- timeserver.jar
```

Das Programm sollte sich nun wie folgt starten lassen:

```
java -p lib -m timeclient/com.timeexample.app.TimeInfoApplication
```

Was muss man tun, um den Start auch folgendermaßen zu ermöglichen?

```
java -p lib -m timeclient
```

Tipp Nutzen Sie das `jar`-Tool und später den Schalter `-main-class`:

```
jar --create --file ...
```

✍ Aufgabe 2 – Abhängigkeiten aufbereiten ✍

Basierend auf den Ergebnissen der Modularisierung aus Aufgabe 1 soll ein Abhängigkeitsgraph erzeugt und visualisiert werden.

Aufgabe 2a: Abhängigkeiten auflisten Bei der Ermittlung der Abhängigkeiten hilft das Tool `jdeps`. Damit sollten Sie in etwa folgende Ausgaben erzeugen:

```
timeclient
 [file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch7/solutions/
      exercise_2/lib/timeclient.jar]
   requires mandated java.base (@9-ea)
   requires timeserver
timeclient -> java.base
timeclient -> timeserver
   com.timeexample.app       -> com.timeexample.services
        timeserver
   com.timeexample.app       -> java.io                    java.base
   com.timeexample.app       -> java.lang                  java.base
   com.timeexample.app      -> java.lang.invoke           java.base
   com.timeexample.app        -> java.time                   java.base
timeserver
```

```
 [file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch7/solutions/
      exercise_2/lib/timeserver.jar]
   requires mandated java.base (@9-ea)
timeserver -> java.base
   com.timeexample.services     -> java.lang              java.base
   com.timeexample.services    -> java.time              java.base
```

Aufgabe 2b: Abhängigkeiten grafisch aufbereiten Zur grafischen Aufbereitung dieser etwas unübersichtlichen Informationen nutzen Sie bitte das Tool `graphviz` (`http://www.graphviz.org/`) in Kombination mit `jdeps`.

Tipp Um die Abhängigkeiten im DOT-Format auszugeben, verwenden Sie den Schalter `-dotoutput`. Damit sollten folgende Ausgaben entstehen:

```
`-- graphs
    |-- summary.dot
    |-- timeclient.dot
    `-- timeserver.dot
```

Bei der Umwandlung solcher DOT-Graphen in ein PNG ist folgendes Kommando nützlich – hier am Beispiel der Datei namens `summary.dot`:

```
dot -Tpng graphs/summary.dot > summary.png
```

Damit sollten Sie folgenden Abhängigkeitsgraphen produzieren:

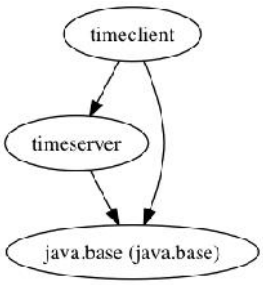

Abbildung 12-1 *Darstellung der Abhängigkeiten*

✍ Aufgabe 3 – JDK-Modul einbinden ✍

Erweitern Sie die Klasse `TimeUtils`, sodass Methodenaufrufe protokolliert werden. Nutzen Sie dazu einen `Logger` aus `java.util.logging` wie folgt:

```
package com.timeexample.services;

import java.time.LocalDateTime;
import java.util.logging.*;

public class TimeUtils
{
    public static LocalDateTime getCurrentTime()
    {
        Logger.getGlobal().info("getCurrentTime() is called");
        return LocalDateTime.now();
    }
}
```

Was fällt beim Kompilieren auf? Wie lässt sich der gemeldete Fehler korrigieren?

Nach der Korrektur sollte der Start mit

```
java -p build -m timeclient/com.timeexample.app.TimeInfoApplication
```

in etwa folgende Ausgaben produzieren:

```
März 10, 2017 8:10:21 NACHM. com.timeexample.services.TimeUtils getCurrentTime
INFORMATION: getCurrentTime() is called
Now: 2017-03-10T20:10:21.136977
```

Tipp Das Kompilieren wird einfacher, wenn man den Multi-Module Build nutzt:

- Für MAC und Linux:

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

- Für Windows mit Powershell:

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

✍ Aufgabe 4 – Executable Runtime erstellen ✍

Das zuvor modularisierte Programmsystem soll als eigenständiges lauffähiges Executable in einem Verzeichnis `runtime_example` bereitgestellt werden.

```
runtime_example
|-- bin
|   |-- java
|   |-- keytool
|   `-- timeclient
...
```

Starten Sie dann das Programm mit `runtime_example/bin/timeclient`, um die Funktionsweise anhand folgender, erwarteter Ausgaben zu prüfen:

```
Nov 26, 2016 2:27:16 PM com.timeexample.services.TimeUtils getCurrentTime
INFORMATION: getCurrentTime() called
Now: 2016-11-26T14:27:16.375422
```

Tipp Denken Sie daran, dass Sie jeweils einzelne JARs etwa wie folgt erzeugen:

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

Das Modul `timeclient` muss zudem als Executable-JAR mit einer Main-Class versehen werden.

Um die Executable Runtime zu generieren, nutzen Sie das Kommando `jlink` sowie die Option `--add-modules timeclient`. Geben Sie auch das JDK im Module-Path an.

✍ Aufgabe 5 – Abhängigkeiten durch Services lösen ✍

Bislang wurden die Programme durch Module strukturiert, besaßen aber noch direkte Abhängigkeiten, also eine starke Kopplung. Für eine losere Kopplung kann man Services nutzen. Dazu bietet Java die Klasse `ServiceLoader` und in Moduldeskriptoren die Schlüsselwörter `uses` und `provides with`.

In der vorherigen Applikation soll das Modul `timeclient` unabhängig vom Modul `timeserver` werden. Dazu kann man ein Interface einführen und dieses in einem separaten Modul `timeservice` bereitstellen:

```
package com.timeexample.spi;

import java.time.LocalDateTime;
import java.util.logging.*;

public interface TimeService
{
    public LocalDateTime getCurrentTime();
}
```

Zudem muss nun die Klasse `TimeUtils` dieses Interface implementieren. Um die Eigenschaft des Service Provider klarer auszudrücken, nennen wir zum einen das Package in `serviceprovider` und zum anderen die Klasse in `CurrentTimeService` um.

Es verbleibt noch der Zugriff auf den Service mithilfe der Klasse `ServiceLoader`:

```
package com.timeexample.app;

import java.util.*;
import com.timeexample.spi.TimeService;

public class TimeInfoApplication
{
    public static void main(final String[] args)
    {
        final Iterator<TimeService> iterator =
                        ServiceLoader.load(TimeService.class).iterator();
        if (iterator.hasNext())
        {
            final TimeService service = iterator.next();

            System.out.println("Now: " + service.getCurrentTime());
        }
    }
}
```

Aufgabe 5a: Untergliedern Sie die Applikation in drei Module In einem ersten Schritt soll die bisherige Applikation in drei Module aufgeteilt und mit folgendem Kommando kompiliert werden:

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

Dabei treten einige Schwierigkeiten auf, was an den noch nicht korrekten Abhängigkeiten zwischen den Modulen liegt.

Aufgabe 5b: Abhängigkeiten in Moduldeskriptoren anpassen Löschen Sie zunächst alle Abhängigkeiten der Module untereinander in den Moduldeskriptoren und fügen Sie dann sukzessive die benötigten `requires`-Anweisungen ein. Korrigieren Sie anschließend die Abhängigkeiten im Kontext der Services und nutzen Sie dazu folgende Bausteine:

```
provides com.timeexample.spi.TimeService
    with com.timeexample.serviceprovider.CurrentTimeService;
```

sowie

```
uses com.timeexample.spi.TimeService;
```

Als Ergebnis des Kompilierens sollte Folgendes entstehen:

```
build
|-- timeclient
|   |-- com
|   |   `-- timeexample
|   |       `-- app
|   |           `-- TimeInfoApplication.class
|   `-- module-info.class
|-- timeserver
|   |-- com
|   |   `-- timeexample
|   |       `-- serviceprovider
|   |           `-- CurrentTimeService.class
|   `-- module-info.class
`-- timeservice
    |-- com
    |   `-- timeexample
    |       `-- spi
    |           `-- TimeService.class
    `-- module-info.class
```

Aufgabe 5c: Lose Kopplung und Ausführbarkeit prüfen Prüfen Sie mit `jdeps`, ob tatsächlich eine mit Services lose gekoppelte, modularisierte Applikation entstanden ist, die folgende Abhängigkeiten aufweist:

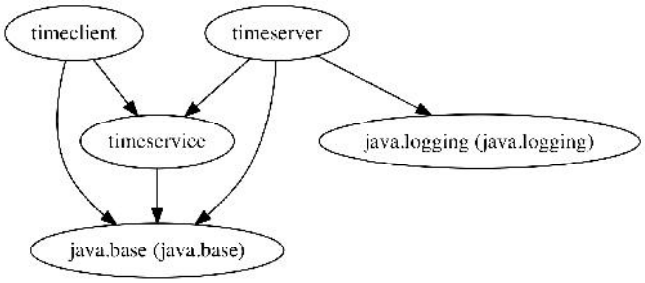

Abbildung 12-2 *Zugriffsmöglichkeiten aus unterschiedlichen Modularten*

Zudem sollte sich das Programm wie folgt starten lassen:

```
java --module-path build -m timeclient/com.timeexample.app.TimeInfoApplication
```

Tipp Nutzen Sie die Tools `jdeps` und `dot` sowie folgende Kommandos:

```
jdeps --module-path build graphs build/*

jdeps --module-path build -dotoutput graphs build/*
dot -Tpng -Gdpi=300 graphs/summary.dot > summary.png
open summary.png
```

✍ Aufgabe 6 – Externe Bibliothek einbinden ✍

Ein herkömmliches Programm mit einer Abhängigkeit auf eine externe Bibliothek soll in eine modularisierte Applikation überführt werden. Das geschieht hier am Beispiel der weitverbreiteten Bibliothek Google Guava in Form der Datei `guava-22.0.jar`. Gegeben sei dazu eine Klasse `UseGuava22FromClassPathExample`, die die Klasse `Joiner` aus Google Guava zum Verketten von Strings nutzt:

```
package com.inden.javaprofi;

import com.google.common.base.Joiner;

public class UseGuava22FromClassPathExample
{
    public static void main(final String[] args)
    {
        final Joiner joiner = Joiner.on(", " ).skipNulls();

        System.out.println(joiner.join("Guava", null, "From", "ClassPath"));
    }
}
```

Dabei ist die gezeigte Klasse in folgender Verzeichnishierarchie definiert:

```
.
|-- externallibs
|   `-- guava-22.0.jar
`-- migration_compatibility_mode
    `-- src
        `-- com
            `-- inden
                `-- javaprofi
                    `-- UseGuava22FromClassPathExample.java
```

Aufgabe 6a: Kompilieren und Starten der Originalapplikation Kompilieren Sie die Applikation mithilfe der Option `-cp` für den `CLASSPATH`:

```
javac -d build -cp ../externallibs/guava-22.0.jar $(find src -name '*.java')
```

Der Start geschieht wie folgt:

```
java -cp build:../externallibs/* com.inden.javaprofi.
     UseGuava22FromClassPathExample
```

Aufgabe 6b: Modularisierte Applikation erstellen Überführen Sie diese einfache Applikation in eine solche, die aus einem Named Application Module namens `modularized_application` und einem Automatic Module besteht.

Die Hauptklasse soll in `UseGuava22AsAutomaticModuleExample` umbenannt werden und nun folgende Konsolenausgabe ausführen:

```
System.out.println(joiner.join("Guava", null, "From", "Automatic"));
```

Das Projekt sollte sich durch folgendes Kommando kompilieren lassen:

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

Dazu müssen Sie sich im Verzeichnis `modularized_application` befinden. Das gilt auch für den Start wie folgt:

```
java -p build:../externallibs -m modularized_application/com.inden.javaprofi.
     UseGuava22AsAutomaticModuleExample
```

Wenn die nachfolgenden Ausgaben produziert werden, haben Sie alles richtig gemacht.

```
Guava, From, Automatic
```

Zudem sollte das Verzeichnis dann folgendermaßen aussehen:

```
.
|-- externallibs
|   `-- guava-22.0.jar
`-- modularized_application
    `-- build
        |-- com
        |   `-- inden
        |       `-- javaprofi
        |           `-- UseGuava22AsAutomaticModuleExample.class
        `-- module-info.class
```