Übungen zur Modularisierung

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Modularisierung und die dortigen Neuerungen überblicksartig näherbringen. Im Anschluss daran sind jeweils einige Übungsaufgaben von den Teilnehmern – idealerweise in Gruppenarbeit – am Rechner zu lösen.

Voraussetzungen

1) Aktuelles JDK 11 installiert
2) Aktuelles Eclipse installiert (Alternativ: NetBeans oder IntelliJ IDEA)

Teilnehmer

- Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 9, 10, 11 und 12 kennenlernen/evaluieren möchten

Kursleitung und Kontakt

Michael Inden
CTO & Leiter ASMIQ Academy
ASMIQ AG, Geerenweg 2, 8048 Zürich
E-Mail: michael.inden@asmiq.ch
Kursangebot: https://asmiq.ch/
Blog: https://jaxenter.de/author/minden

Modularisierung – Project Jigsaw

Nachfolgend wird ein einfaches Programm in mehreren Iterationen in eine modularisierte Applikation überführt und mit verschiedenen Erweiterungen versehen.

Aufgabe 1 – Applikation in zwei Module aufspalten

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.

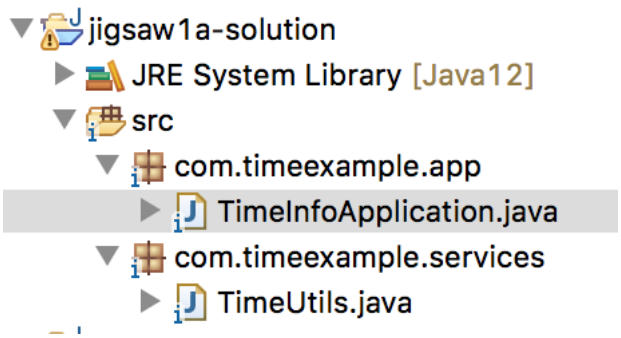
```
package com.timeexample;

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`.



Aufgabe 1b: Applikation modularisieren
Spalte in diesem Schritt die Applikation in die 2 Module mit dem Namen `timeclient` und `timeserver` auf. Verwenden Sie dazu in Eclipse jeweils eigene Projekte und in IntelliJ bietet sich ein hierarchisches Projekt mit zwei Modulen an.

Überführen Sie dann die Packages passend in die Module. Nutzen Sie jeweils eine `module-info.java`-Datei zur Moduldefinition, in der die Abhängigkeiten korrekt spezifiziert sind. Bedenken Sie auch die Auswirkungen der Modulabhängigkeiten und passen die Einstellungen in der IDE geeignet an. Starten Sie dann die modularisierte Applikation.

Tipp 1: Beim Kompilieren könnte es zu Fehlern kommen:

Der Funktionalität zum Erzeugen eines Moduldeskriptors ist in Eclipse etwas versteckt im Kontextmenü des Projekts unter `Configure > Create` module-info.java.

Tipp 2: Beim Kompilieren könnte es zu Fehlern kommen:

Prüfen Sie die Moduldeskriptoren auf Abhängigkeiten und Freigaben, wie z. B. `requires XYZ` und `exports com.abc.def`. Dazu bieten die IDEs gewisse Quick Fixes.

Tipp 3: Nutzen Sie den Module-Path beim Starten aus dem Hauptverzeichnis jigsaw1b und verweisen Sie auf beide Projekte:

```
java –p ./TimeServer/bin:./TimeClient/bin \
     –m timeclient/com.timeexample.app.TimeInfoApplication
```

Aufgabe 1c: Modulare JARs erstellen
Erstellen Sie aus den kompilierten Sourcen der einzelnen Module bzw. Subverzeichnisse zwei modulare JARs und legen Sie dazu ein Verzeichnis `libs` im jeweiligen Hauptverzeichnis an, das als Ziel für die Module dient. Kopiere später dann das erzeugte jar aus dem `TimeServer` in den `TimeClient`. Das Verzeichnis sollte wie folgt aussehen:

```
├── TimeClient
│   ├── bin
│   │   ├── com
│   │   │   └── timeexample
│   │   │       └── app
│   │   │           └── TimeInfoApplication.class
│   │   └── module–info.class
│   …
│
└── TimeServer
    ├── bin
    │   ├── com
    │   │   └── timeexample
    │   │       └── services
    │   │           └── TimeUtils.class
    │   └── module–info.class
```

Als Ergebnis wird dann folgende Ergänzung erwartet:

```
├── libs
│   ├── timeclient.jar
│   └── timeserver.jar
```

Das Programm sollte sich nun wie folgt starten lassen:

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

ACHTUNG / ACHTUNG / ACHTUNG:
Aufgrund eines Fehlers im JAR-Tool ist das Ganze derzeit nicht möglich und es kommt zu folgender Fehlermeldung:

```
Error occurred during initialization of boot layer
java.lang.LayerInstantiationException: Package com in both
module timeserver and module timeclient
```

Man kann aber einen Gradle/Maven-Build verwenden und die entstehenden JARs in einem Verzeichnis sammeln.

Tipp: Beim Erzeugen von JARs hilft folgendes Kommando In den jeweiligen Projekthauptverzeichnissen:

```
> jar --create --file libs/<myjarname>.jar -C bin .
```

Tipp: Beim Kopieren der JARs hilft folgendes Kommando:

```
> mv libs/timeserver.jar ../TimeClient1/libs
```

ABHILFE GRADLE-BUILD

Nutzen Sie als Abhilfe einen gradle-Build und eine Datei `build.gradle` bestehend aus folgenden Zeilen

```
apply plugin: 'java'

sourceCompatibility=11
targetCompatibility=11
```

im Projekthauptverzeichnis sowie das Kommando `gradle clean assemble`. Für das Projekt `timeclient` benötigt es noch die Angabe der Abhängigkeiten nach den obigen Angaben:

```
tasks.withType(JavaCompile) {
    options.compilerArgs +=
 ["--module-path",
  "${buildDir}/../../TimeServer1/build/libs"]
}
```

Aufgabe 2 – Abhängigkeiten aufbereiten

Basierend auf der Aufspaltung der Applikation in zwei Module gemäß Aufgabe 1 soll nun ein Abhängigkeitsgraph erzeugt und visualisiert werden. Das Ganze wir erleichtert, wenn man sich die durch gradle entstehenden JAR-Dateien in ein separates Verzeichnis `libs` kopiert.

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

```
timeclient
 [file:///Users/michael.inden/eclipse-
workspace/TimeClient2/libs/TimeClient2.jar]
   requires mandated java.base (@12)
   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/michael.inden/eclipse-
workspace/TimeClient2/libs/TimeServer2.jar]
   requires mandated java.base (@12)
   requires java.logging (@12)
timeserver -> java.base
   com.timeexample.services         -> java.lang            java.base
   com.timeexample.services         -> java.time
java.base
java
```

Das ist allerdings recht unübersichtlich. Wie erhält man eine Zusammenfassung ähnlich zu folgender?

```
timeclient -> java.base
timeclient -> timeserver
timeserver -> 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` und dem Schalter `-dotoutput`.

```
jdeps –dotoutput graphs libs/*.jar
```

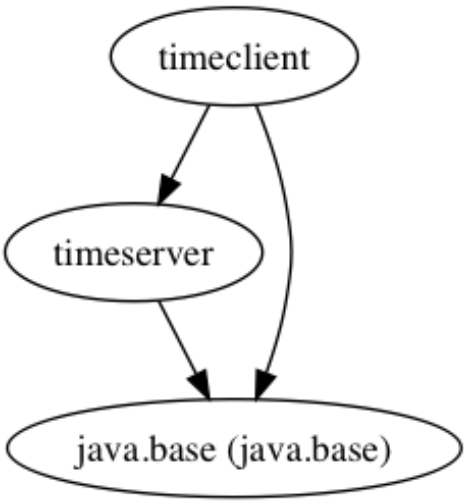
Damit sollten folgende Dateien entstehen:

```
├── graphs
│   ├── summary.dot
│   ├── timeclient.dot
│   └── timeserver.dot
```

Bei der Umwandlung von 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:



Aufgabe 3 – Ein Modul aus dem JDK 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.Logger;

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 einfach korrigieren?

Aufgabe 4 – Executable Runtime erstellen

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

Aufgabe 4a: JARs erzeugen
Zunächst müssen jeweils einzelne JARs etwa wie folgt erzeugt werden, dazu würde man normalerweise Folgendes nutzen:

```
jar --create --file libs/xyz.jar -C bin/XYZ .
```

Aufgrund eines Fehlers im JAR-Tool derzeit nicht möglich! Man kann aber einen Gradle/Maven-Build verwenden und die entstehenden JARs in einem Verzeichnis libs sammeln.

Aufgabe 4b: JAVA_HOME auf Runtime korrekt setzen
Prüfen Sie die Umgebungsvariable JAVA_HOME und setzen Sie diese – sofern nötig – auf ein gewünschtes JDK, etwa JDK 11:

```
export JAVA_HOME=`/usr/libexec/java_home -v "Java SE 11.0.1"`
export PATH=$JAVA_HOME/bin:$PATH
```

Aufgabe 4c: Executable erzeugen
Erstellen Sie mithilfe von `jlink` ein Executable im Verzeichnis `runtime_example`: Das Ergebnis sollte in etwa so aussehen:

```
runtime_example/
├── bin
│   ├── MYEXE
│   ├── java
│   └── keytool
├── conf
│   ├── logging.properties
```

Starten Sie dann das Programm, um die Funktionsweise zu prüfen:

```
> ./runtime_example/bin/MYEXECUTABLE
Apr 19, 2019 2:26:58 PM com.timeexample.services.TimeUtils
getCurrentTime
INFO: getCurrentTime() is called()
Now: 2019-04-19T14:26:58.
```

Tipp 1: 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.

Tipp 2: Denken Sie an die Optionen `--launcher` und `--output`.

Tipp 3: Versuchen Sie es mal in etwa wie folgt:

```
jlink -p $JAVA_HOME/jmods:Build/libs:../TimeServer4/build/libs \
      --add-modules timeclient \
      --launcher=MYEXE=timeclient/com.timeexample.app.TimeInfoApplication \
      --output runtime_example
```

Aufgabe 4d: Wie gross bzw. klein ist das Verzeichnis?
Nutzen Sie den Windows Explorer, den Mac Finder oder aber wieder die Kommandozeile:

```
du -sh runtime_example/
```

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:

Aufgabe 5a: Ein neues Modul TimeServices erzeugen
Folgendes Interface dient zur Entkopplung von Client und Server:

```
package com.timeexample.spi;

import java.time.LocalDateTime;

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

Erstellen Sie dazu zugehörige Modul und führen einen ersten Build aus.

Aufgabe 5b: Umstellung der beiden bestehenden Module

Zunächst muss die Klasse `TimeUtils` das obige Interface implementieren und auf den Service mithilfe der Klasse `ServiceLoader` aus dem JDK zugreifen. Nutzen Sie folgende Implementierung:

```
package com.timeexample.app;

import java.util.Iterator;
import java.util.Optional;
import java.util.ServiceLoader;

import com.timeexample.spi.TimeService;

public class TimeInfoApplication
{
	public static void main(final String[] args)
	{
		final Optional<TimeService> optService =
					lookup(TimeService.class);
		optService.ifPresentOrElse(service ->
		{
			System.out.println("Service: " + service.getClass());
			System.out.println("Now: " + service.getCurrentTime());
		},
		() -> System.err.println("No service provider found!"));
	}

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

Ebenfalls muss die Klasse `TimeUtils` wie folgt angepasst werden:

```
package com.timeexample.services;

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

import com.timeexample.spi.TimeService;

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

Ignorieren Sie zunächst die Kompilierfehler in beiden Modulen

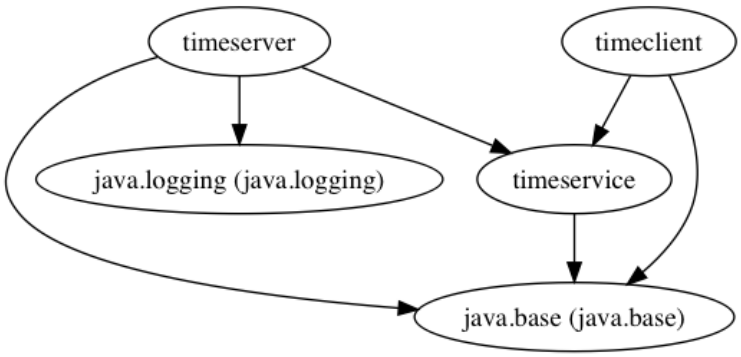
Aufgabe 5c: Abhängigkeiten in den Moduldeskriptoren anpassen
Löschen Sie zunächst alle Abhängigkeiten der Module untereinander in den Moduldeskriptoren und fügen Sie dann sukzessive die benötigten `requires`-Anweisungen sowie die Abhängigkeiten zwischen den Eclipse-Projekten wieder ein. Korrigieren Sie anschließend die Abhängigkeiten im Kontext der Services. Beginnen Sie mit dem Modul `TimeUtils` und nutzen Sie dazu folgende Bausteine:

```
provides com.timeexample.spi.TimeService
with com.timeexample.services.TimeUtils;
```

Sowie:

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

Aufgabe 5d: Alle Module kompilieren und lose Kopplung prüfen
Kompilieren Sie jedes Modul für sich und passen Sie ggf. die Build-Dateien entsprechend der neuen Abhängigkeiten an. Sammeln Sie wieder alle JARs in einem Verzeichnis `libs` im `TimeClient`-Hauptverzeichnis. Es sollte in folgender Abhängigkeitsgraph generiert werden:



Tipp: Nutzen Sie Abwandlungen des folgenden Kommandos zum Befüllen des `libs`-Ordners:

```
cp build/libs/*.jar libs
```

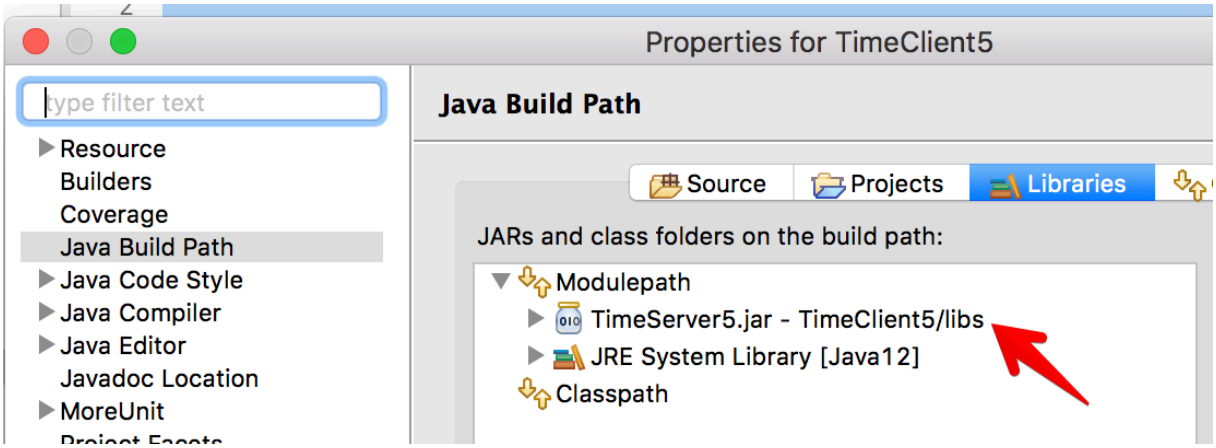
Aufgabe 5e: Ausführen mit Services
Trotz der offensichtlich nicht mehr vorhandenen direkten Abhängigkeit zwischen den Modulen `timeclient` und `timeserver` sollte die Applikation trotzdem weiterhin funktionieren. Starten Sie diese.

```
Service: class com.timeexample.services.TimeUtils
Apr. 19, 2019 6:08:08 NACHM. com.timeexample.services.TimeUtils
getCurrentTime
INFO: getCurrentTime() is called()
Now: 2019-04-19T18:08:08.913864
```

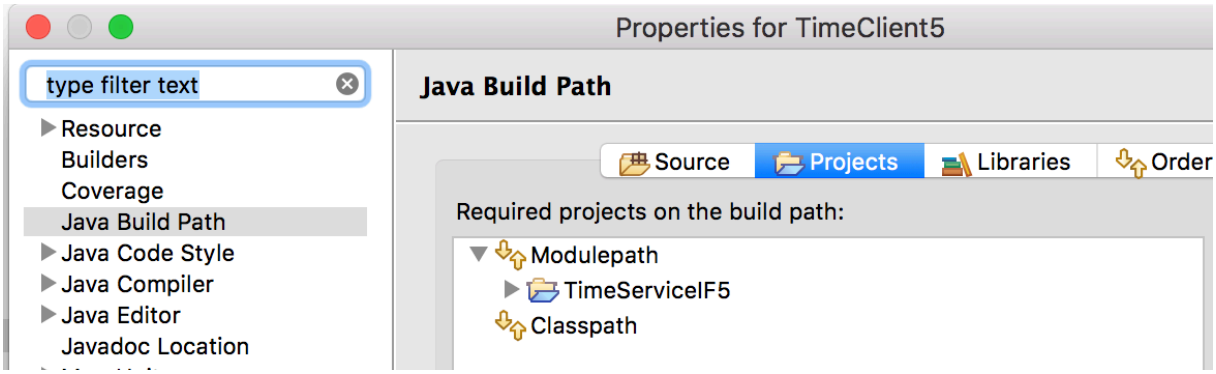
In Eclipse könnten Sie sich beim Applikationsstart wundern: Es wird nur eine Fehlermeldung ausgegeben:

```
<terminated> TimeInfoApplication (3) [Java
No service provider found!
```

Wie kommt das? Nach kurzem Nachdenken vermuten Sie wohl eine fehlende Abhängigkeit im `TimeClient`-Projekt. Dieses benötigt zwar zur Kompilierzeit keine Abhängigkeit, zur Laufzeit muss es aber auf die Implementierung zugreifen können. Ergänzen Sie folgendes und starten erneut.



Prüfen Sie, ob Sie nur vom Interface abhängig sind:



Aufgabe 6 – Externe Bibliothek einbinden

Nehmen wir an, wir würden gerne in unserem Applikationsmodul einige Funktionalität aus Google Guava einsetzen, aktuell der Version guava-27.0.1-jre.jar. Nehmen wir an, wir hätten die Bibliothek schon in einem Ordner `externallibs` zur Verfügung, damit es für Eclipse einfachen ist. Später schauen wir auf Gradle und Dependencies beim Build.

Beginnen wir mit dem Kompatibilitätsmodus, bei dem man einem Modul einen CLASSPATH mitgeben kann:

```
javac -d build -cp ../externallibs/guava-23.0.jar \
      -p ../TimeServiceIF5/build/libs $(find src -name '*.java')
```

Das ist zwar möglich, oftmals ist es aber sinnvoller, das JAR als Automatic Module einzubinden, dazu ist lediglich das JAR-File in den Module Path aufzunehmen:

```
javac -d build -p ../externallibs/guava-
23.0.jar:../TimeServiceIF5/build/libs $(find src -name '*.java')
```

Dazu muss der Moduldeskriptor wie folgt angepasst werden:

```
    requires guava;
```

⇨ Name of automatic module 'guava' is unstable, it is derived from the module's file

Guava 27 lässt sich durch eine spezielle Angabe im Manifest als Automatic Modul mit spezifischem Namen einbinden:

```
    Automatic-Module-Name: com.google.common
```

```
    requires com.google.common;
```

Der Gradle-Build muss wie folgt angepasst werden:

```
// bisschen getrickst aber ..
tasks.withType(JavaCompile) {
    options.compilerArgs +=
["--module-path",
"${buildDir}/../../TimeServiceIF5/build/libs:${buildDir}/../externallibs"]
}

repositories
{
      jcenter()
}

dependencies
{
      compile group: 'com.google.guava', name: 'guava', version: '27.1-jre'
}
```

Probieren Sie einen Applikationsstart mit

```
    java -p libs:externallibs/guava-27.0.1-jre.jar \
         -m timeclient/com.timeexample.app.TimeInfoApplication
```

Das sollte nun Folgendes ausgeben:

```
Service: class com.timeexample.services.TimeUtils
Apr. 19, 2019 6:50:36 NACHM. com.timeexample.services.TimeUtils
getCurrentTime
INFO: getCurrentTime() is called()
Now: 2019-04-19T18:50:36.903890
Guava, From, ModulePath
```

YOU HAVE SUCCESFULLY FINSHED MODULARIZATION ☺
