Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Vorlesung Objektorientierte
Komponenten-Architekturen
(OOKA)
WS 2020 / 2021

Prof. Dr. Sascha Alda

19. November 2020

Übungen zur Vorlesung
Objektorientierte Komponenten-Architekturen

WS 2020 / 2021

Übungsblatt Nr. 2
(Abgabe bis: Mittwoch, den 02. Dezember 2020, **09:00 Uhr**)

Aufgabe 1 (Laufzeitumgebung für Komponenten):

Entwickeln sie eine eigene JVM-Laufzeitumgebung für Software-Komponenten ☺

Wenden sie dabei ihre aus der Vorlesung gewonnen Erkenntnisse für den Aufbau einer Komponenten-Laufzeitumgebung und eines Komponentenmodells an. Bei der Gestaltung haben sie genügend Freiheiten, es *müssen* jedoch folgende funktionale Anforderungen erfüllt sein:

FA1: Der Component Assembler muss die Laufzeitumgebung *starten* können.

FA2: Der Component Assembler muss die Laufzeitumgebung *stoppen* können.

FA3: Der Component Assembler muss neue Komponenten aus einem lokalen Verzeichnis des Rechners in die Laufzeitumgebung einsetzen (*deployen*) können.

FA4: Der Component Assembler muss eingesetzte Komponenten in der Laufzeitumgebung *ausführen* können. Eine Start-Methode sollte dabei mit Hilfe einer Annotation im Rahmen eines Komponentenmodells definiert werden.

FA5: Der Component Assembler soll mehrere Komponenten gleichzeitig (parallel) ausführen können.

FA6: Der Component Assembler muss in der Lage sein, die Status der *aktuell* eingesetzten Komponenten über die Laufzeitumgebung abzurufen. Ein Status sollte *pro* Komponente folgendes beinhalten: Laufende Identifikationsnummer, Name, Zustand.

FA7: Der Component Assembler muss Komponenten in der Laufzeitumgebung *stoppen* können. Auch hierzu sollte das Komponentenmodell eine „entsprechende“ Stop-Methode bereitstellen.

FA8: Der Component Assembler muss Komponenten aus der Laufzeitumgebung löschen können.

Wenden sie dabei ihre aus der Vorlesung gewonnen Erkenntnisse für den Aufbau einer Komponenten-Laufzeitumgebung und eines Komponentenmodells an (*gesamtes* Kapitel 2, hier ggf. schon vorarbeiten). Bei der Gestaltung haben sie aber auch genügend Freiheiten

(z.B. bezüglich der Gestaltung der UI), es *müssen* jedoch die obigen funktionalen Anforderungen erfüllt werden.

Beachten sie zudem folgende Teilaufgaben:

1.)
Ein Komponentenmodell beschreibt die Struktur einer Komponente. Folgende strukturelle Eigenschaften sollte eine Komponente besitzen:

- Eine „deploybare“ Komponente entspricht einem .jar-File,
- Ein .jar-File kann wiederum eine beliebige Menge aus Klassen enthalten kann. In einem .jar-File *muss genau* eine Klasse mit einer annotierten start-Methode sowie einer annotierten stop-Methode enthalten sein (die „Starting Class“)
- Klassen in einem .jar-File *sollten* in Packages angeordnet sein.
- Eine Komponente *kann* ein Meta-File besitzen, z.B. zur Beschreibung von Abhängigkeiten ([optional]!).
- Eine Komponente *kann* ein lib-Verzeichnis zur Aufnahme von abhängigen Libraries (z.B. JDBC-Treiber) besitzen ([*optional*]!). Ratschlag: bitte diese Libraries eher in den Classpath der Laufzeitumgebung ablegen. Das Auslesen von Libraries aus dem lib-Verzeichnis kann „tückisch“ sein.

2.)
Als deploybare Beispiel-Komponente *können* sie die in der Übung Nr. 1 entwickelte Komponente eines Buchungssystems verwenden (.jar File als Package-Format!). Diese Komponente muss dann entsprechend dem zu entwickelnden Komponentenmodell angepasst werden (z.B. um eine Start-Methode in einer *internen* Klasse Client, welche die Abfragen über das Provided Interface initiiert und den Cache setzt). Zugehörige Libraries (z.B. die JDBC-Treiber-Klassen) sollten sie in den Classpath der Laufzeitumgebung ablegen (siehe Anmerkung oben). Für Demo-Zwecke können sie aber auch zu Beginn eine einfache Komponente zusätzlich implementieren und deployen (z.B. für den Test bzw. Vorführung der parallelen Ausführbarkeit von Komponenten).

3.)
Modellieren sie zudem ein UML-basiertes Zustandsmodell für ihre Anwendung, um die möglichen Zustände darzustellen, die ihre Komponenten einnehmen können. Berücksichtigen sie auch mögliche Zustandsübergänge inklusive annotierter Aktionen (Methoden) aus dem Kontext einer Komponente, die bei einem Zustandsübergang ggf. ausgeführt werden.

4.)
Testen sie den Classloader-Mechanismus ihrer Laufzeitumgebung angemessen mit Hilfe diverser JUnit-Tests. Sie sollten hierzu JUnit 4 oder JUnit 5 verwenden. Zeigen sie mittels des Tests auch, dass ihr Classloader ihre Klassen aus den Komponenten *isoliert* lädt. Weitere Hinweise dazu unten (Link auf Artikel in Jax-Center).

5.)
Dokumentieren sie kurz, welche Kriterien aus dem Rahmenwerk nach Crnkovic ihr Komponentenmodell erfüllt. Konzentrieren sie sich dabei auf die Bereiche „Lifecycle“ und „Construction“ (vgl. Kapitel 2, Folien 24-25).

Quellen für die Entwicklung

Folgende Quelle könnte ihnen bei der Programmierung der Laufzeitumgebung hilfreich sein (Last Access: 18.11.2020)

Grundlagen zum Thema Java Classloading:
http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_11_002.htm

Java Reflection (API) zur Analyse einer Klassenstruktur:
http://www.java2s.com/Tutorial/Java/0125__Reflection/Demonstrateshowtogetspecificmethodinformation.htm

Für solche komplexen Java-Operationen sei auch wiederholt auf die (stets!!) guten Dokumentationen und Hinweise auf der Seite stackoverflow.com verwiesen. Beispiel: Laden aller Klassen aus einem .jar-File (Last Access: 18.11.2020):
https://stackoverflow.com/questions/11016092/how-to-load-classes-at-runtime-from-a-folder-or-jar

Hinweis: diese Quelle enthält einige gute (!) Codes zum Laden und Bereitstellen von Klassen aus einem .jar-File, die ein guter Ausgangspunkt für die Entwicklung der Laufzeitumgebung (insbesondere des Lade-Mechanismus) darstellen!

Auch eine gute Quelle für Class Loading:
https://www.philipphauer.de/study/se/classloader.php

Hier eine interessante Quelle über das „Problem“ von zwei gleichen Klassen, die von zwei Classloader geladen werden, um eine Isolation von Klassen zu erhalten (betr. Dependency Injection, vgl. auch Teilaufgabe Nr. 3):
https://jaxenter.de/aus-der-java-trickkiste-class-loading-20271

Grundlagen zur Implementierung einer einfachen Ein- und Ausgabe_
http://openbook.galileocomputing.de/javainsel9/javainsel_11_004.htm#mj519f885647f50baccfda39badd91c631

Zur Entwicklung der Kommandos der Laufzeitumgebung empfehle ich die Verwendung des *Command Pattern*:
Quelle: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Pattern. 1. Auflage. Addison-Wesley, 1996

Grundlagen zur Thread-Programmierung:
https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Weitere Hinweise zur Implementierung finden sie in der Tabelle, die wir heute in der Vorlesung (19.11.2020) zusammengetragen haben. Einen Ausdruck davon finden sie auf LEA (PDF).

Ich empfehle im Allgemeinem die Umsetzung der Laufzeitumgebung mit der Programmiersprache Java (ab Version 8). Die Besprechung der Lösungen und auch eine Musterlösung (wird *auszugsweise* bei *dringendem* Bedarf bereitgestellt) erfolgt auf Basis der Sprache Java. Für die finale Programmierung im Rahmen eines Semesterprojekts akzeptiere ich auch alternative JVM-basierte Sprachen wie z.B. Kotlin oder Scala. Dies sollte aber nur von in den Programmiersprachen erfahrenen (oder interessierten) Entwicklern / Studierenden angegangen werden!