Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Vorlesung Objektorientierte
Komponenten-Architekturen
(OOKA)
SS 2025

Prof. Dr. Sascha Alda

31. März 2025

# Übungen zur Vorlesung
# Objektorientierte Komponenten-Architekturen

## SS 2025

**Übungsblatt Nr. 1**
(Abgabe: 15. April 2025, **13:00 Uhr via LEA**)

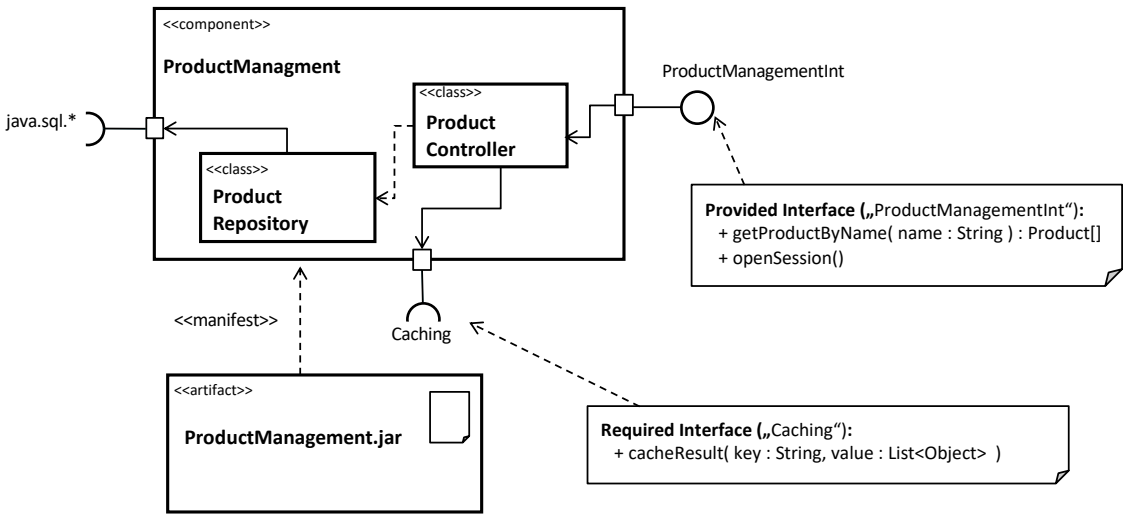
**Aufgabe 1 (Source Code zu Komponenten):**

Die Java-basierte Umsetzung des Komponentendiagramms (vgl. Folie 28 bzw. 29, Kapitel 1) entspricht *einer* Möglichkeit der Abbildung des Modells auf einen äquivalenten Source Code. In dieser Aufgabe sollten sie den Source Code optimieren, in dem sie das *Port*-Konzept aus diesem Diagrammtyp anwenden und exemplarisch implementieren. Beantworten sie folgende Fragen:

- Welche Aufgabe haben Ports in einem Komponentendiagramm? Recherchieren sie dieses Konzept anhand der Quelle von (Rupp, 2012), Kapitel 9.1.3 und 9.4.3
- Wie könnte man deren Aufgabe für eine Komponente in Java implementieren? (Siehe dazu auch die Anforderung *FA0*)
- Wie können benötigte bzw. angebotene Interfaces injiziert bzw. entnommen werden?

Rupp et al.: *UML2 Glasklar. Praxiswissen für die UML-Modellierung*. 5. Auflage. Hanser, 2012. (in der Bibliothek mehrfach vorhanden). Eine digitale Kopie befindet sich auf dem OOKA-Literatur-Repository:

https://git.fslab.de/salda2m/ooka#modellierung-von-software-architekturen



Sie haben nun die Aufgabe, das obige Modell eines ProductManagement-Systems auf Grundlage eines Port-Konzepts zu implementieren! Verwenden sie dazu die Sprache Java (ab) Version 19. Verwenden sie auch eine entsprechende IDE ihrer Wahl (meine Empfehlung: IntelliJ). Verwenden Sie dazu eine H2-Datenbank, die Sie z.B. über eine Maven-Dependency (Empfehlung) zu Ihrem Projekt hinzufügen können.

Eine erste gute Basis-Implementierung mit hilfreichen Kommentaren können Sie unter folgendem Repository runterladen (bzw. clonen; Codes liegen auf dem Master-Branch):

https://git.fslab.de/salda2m/okaaue1

Implementieren Sie nun folgende funktionalen Anforderungen. Die dazu gehörigen Fragen bitte kurz in einem Text-Dokument adressieren und beantworten:

*FA0*: Implementieren sie das Port-Konzept nach den Vorgaben bzw. Überlegungen gemäß (Rupp, 2012). Welches Design Pattern sollte hier verwendet werden, um die notwendige Delegation zwischen internen und externen Verhalten zu realisieren?

*FA1*: Sie finden eine erste Implementierung des Interface `ProductManagementInt` auf dem o.g. GitLab-Repository.

Beachten Sie auf jeden Fall die Spezifikation zum Methodenaufruf in den Kommentar des Interfaces. Muss das Interface `ProductManagementInt` ggf. noch um weitere Methoden erweitert werden? Gibt es eine dedizierte Reihenfolge beim Aufruf der Methoden des Interfaces? Implementieren Sie die Lifecycle-Methoden rudimentär anhand eines einfachen (!) Zustandsmodells sowie unter Beachtung der Kommentare (Hinweis auch hier: in den Lifecycle-Methoden sollten Sie das Öffnen und Schließen einer Datenbank-Verbindung implementieren).

FA2: Implementieren Sie Komponente `ProductManagement`. Binden Sie dazu eine externe H2-Datenbank ein, die man über Maven als Dependency einbinden kann (siehe Beispiel-Code in dem obigen Repository). Verwenden Sie für den Zugriff auf die Datenbank die Schnittstellen aus JDBC. Erste Code-Beispiele finden Sie in dem Junit-Test aus dem Repository (`ConnectionTest`). Erweitern Sie den Junit-Test, um die Komponente hinreichend zu testen. Ergänzen Sie vor allem den Round-Trip-Test.

*FA2*: Die Komponente `ProductManagement` benötigt ferner eine Referenz vom Typ `Caching`, mit der die interne Klasse `ProductController` die Ergebnisse in einem Cache zwischenspeichern kann. Von außerhalb der Komponente muss also über einen externen Client eine entsprechende Referenz erzeugt werden und über den Port *injiziert* werden. Ist die Schnittstelle `Caching` hinreichend modelliert oder fehlen auch hier Methoden? Implementieren sie die Implementierung eines konkreten Cache *rudimentär* (z.B. über eine HashMap oder Liste aus `java.util`).

*FA3*: Überlegen sie auch einen Mechanismus, damit `ProductController` stets zumindest scheinbar ohne Probleme (z.B. keine NullPointer-Exceptions) auf den Cache zugreifen kann, auch wenn *keine* konkrete Referenz gesetzt ist. Ein etwaiges Fehlerhandling darf dabei *nicht* von der Klasse `ProductController` übernommen werden.

*FA4*: Realisieren sie zudem eine Logging-Funktionalität, mit der die Zugriffe auf das Interface `ProductManagementInt` geloggt werden. Eine Ausgabe sollte wie folgt sein:

```
01.04.25 21:22: Zugriff auf ProductManagement über Methode getProductByName. Suchwort: Motor
```

Auch das Logging ist eine Querschnittsfunktionalität, die *nicht* in der Klasse `ProductController` enthalten sein soll.

*FA5:* Ihre gesamten Entwicklungen sollen dann als „deploybare“ Komponente im Format `.jar` exportiert werden (z.B. über *ein passendes* JAR-Plugin in Maven, das über die Lifecycle-Phase `package` ausgeführt werden kann). Testen sie ihre Entwicklungen hinreichend mit einem externen Client, der in einem neuen Projekt liegt, also nicht Teil der deploybaren Komponente ist. In dem Client sollte dann auch eine Instanz einer eigenen Cache-Implementierung eingesetzt werden. Achten Sie auch darauf, dass der Datenbank-Treiber in dem JAR-File vorhanden ist (was passiert, wenn dies nicht der Fall ist?). Siehe dazu auch den Hinweis im POM-File des Beispielprojekts auf GitLab. Der Client ruft einfach die Methoden der Komponenten im Rahmen eines beispielhaften Szenarios auf. Ein UI, GUI oder CLI ist nicht notwendig.

*R1:* Modellieren Sie die resultierenden Klassen und die Abhängigkeiten ihrer gesamten Software als ein Klassendiagramm nach der UML. Betrachten sie auch das Komponenten-Diagramm: Um welche externen Komponenten könnte man dieses erweitern? Falls Sie eine Erweiterung sehen, dann modellieren die Erweiterungen entsprechend. Modellieren Sie auch das resultierende Zustandsmodell (vgl. *FA1*) mit einem einfachen (!) UML-Zustandsdiagramm.

Bitte laden sie als Ergebnis die Source Codes, das .jar File sowie kurze Antworten zu den obigen Fragen (wenn vorhanden) auf LEA hoch. Als Alternative können sie den Source Code auf ein eigenes öffentliches GitHub-Repository bereitstellen und den Link dazu auf LEA hochladen (z.B. in einer readme-Datei). Die Antworten zu den Fragen sollten als PDF bereitgestellt werden.

**Allgemeine Hinweise:**

Für die Modellierung mit UML empfehle ich folgendes Tool:

- Draw.io

Schlankes browser-basiertes Tool, keine Installation auf ihrem Rechner notwendig! Abspeicherung der Dokumente in verschiedenen Formen möglich (Lokal, Cloud). Läuft nativ ohne Plugin auf allen gängigen Browsern:

https://app.diagrams.net/

Für die anstehenden Entwicklungen empfehle ich das Tool IntelliJ. Hier empfiehlt sich der Download der Ultimate-Version, die als registrierter Student kostenlos bezogen werden kann! Mit dieser Ultimate-Version können u.a. auch Java EE Anwendungen entwickelt werden. Auch für moderne Entwicklungsstandards wie Microservices, Docker, Maven usw. bietet IntelliJ einen guten Support. Unübertroffen ist die Auto Completion Funktion, welches IntelliJ recht populär gemacht hat. Auch eine KI-Erweiterung kann mit dem Tool „GitHub CoPilot“ einfach eingerichtet werden:

https://www.jetbrains.com/idea/

Sollten sie noch Probleme haben mit der IDE oder auch mit der Integration von GitHub, so verweise ich gerne auf eine eigene Tutorien-Reihe.

Teil 1: Installation IntelliJ und Entwicklung eines Java-Projekts mit JUnit5 –
https://www.youtube.com/watch?v=TNtRpkdW64s

Teil 2: Clone eines GitHub-Repository mit IntelliJ –
https://www.youtube.com/watch?v=5nr4c3pwu3g

(*Hinweis*: in diesem Tutorium wird ein anderes GitHub-Repro verwendet; bitte den o.g. Link zu dem OOKA-Repro verwenden. Bitte auch den Maven-Support aktivieren.)

Teil 3: Push von Source Code auf ein GitHub-Repository mit IntelliJ
https://www.youtube.com/watch?v=PbGiYUR9q0A