# Praktikumsaufgabe zur Qualitätssicherung

In diesem Praktikum übernehmen Sie die Entwicklung eines bestehenden Systems namens "Parcel Services (kurz: Parcer)". In der Vorlesung wurden die architektonischen und entwicklungsprozessualen Unzulänglichkeiten von Parcer bereits beschrieben (s. Folie "Fallstudie"). Im Folgenden werden Sie Schritte zur Verbesserung unternehmen. Im Einzelnen werden Sie

- den Build um weitere Qualitätssicherungs-Schritte erweitern.
- (optional) die monolithische Struktur von Parcer ein Stückweit aufbrechen, indem Sie das Parcer-Maven-Modul in mehrere Maven-Module zerlegen,

In manchen Aufgaben werden Verständnisfragen gestellt. Bitte machen Sie sich bei der Bearbeitung der Aufgaben entsprechende Notizen.

| **IMPORTANT** | Als Abgabe zählt bei diesem Praktikumstermin der mit `abgabe` getaggte Commit in Ihrem `parcer`-Remote-Repository. |
|---|---|

## Parcer im Überblick

### Anwendungsfälle

Parcer ist ein Versandlogistik-System, das Versandkunden eines Logistikdienstleisters nutzen, um Pakete für den Versand vorzubereiten. Es unterstützt die Anwendungsfälle wie in [Use-Cases von Parcer] dargestellt.

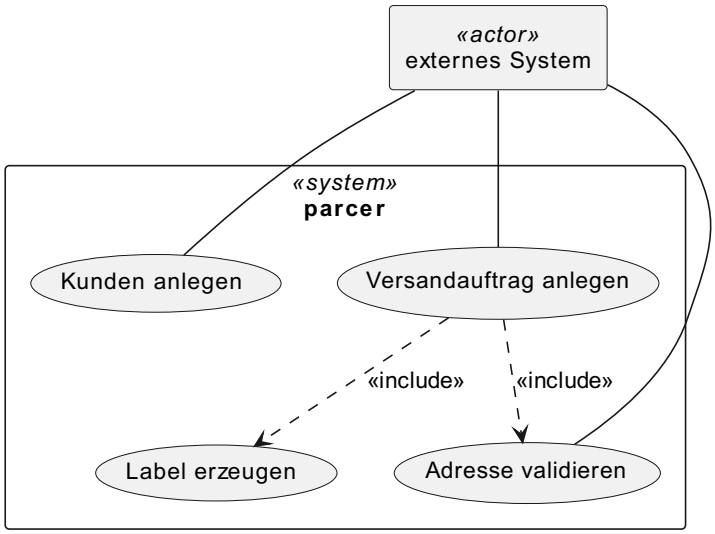


*Figure 1. Use-Cases von Parcer*

Parcer exponiert WebAPIs für externe Systeme, die beispielsweise Warenwirtschaftssysteme oder WebShop-Systeme von Versandkunden aufrufen können.

| **NOTE** | Parcer ist real existierenden Systemen entlehnt. Der Code- und Funktionsumfang von Parcer ist aus didaktischen Gründen und aus Aufwandsgründen natürlich um mehrere Größenordnungen kleiner als bei den real existierenden Systemen. |
|---|---|

### Monolithische Ausgangs-Architektur

### Java-Package-Struktur

Die Softwarearchitektur besteht aus mehreren Komponenten, die in Java als Java-Packages realisiert sind. Diese Packages und ihre Abhängigkeiten sind in [Package und Dependencies] dargestellt.

*Packages und Dependencies*

image

Die Verantwortlichkeiten der Komponenten sind dabei wie folgt:

- Package `controllers`:
  - Klassen in diesem Paket besitzen Methoden, die bei Requests an die WebAPIs aufgerufen werden und denen eventuelle Daten im Request übergeben werden. Diese Methoden delegieren die Verarbeitung an Klassen im Package services.
  - Beispiel: Der Methode `CustomerController.createCustomer(Customer)` wird ein `Customer`-Objekt übergeben. Dieses reicht die Methode weiter an `CustomerService.createCustomer(Customer)` .
- Package `domain`
  - Fachliche Daten werden in Objekten von Klassen gehalten, die in Package `domain` definiert sind. Die Erzeugung dieser Objekte geschieht mitunter automatisch, beispielsweise als Formalparameter (Eingabeparameter) beim Aufruf von Methoden im Package `controllers`.
  - Beispiel: Die Klasse `Customer` repräsentiert einen Kunden, der eine eindeutige `id`, einen `name` und Menge von zugeordneten `Shipment`s hat.
- Package `resources`:
  - WebAPI-Responses werden aus Objekten serialisiert, deren Klassen im Package `resources` zu finden sind. Diese Klassen kapseln die aus `domain`, erben jedoch von einer Spring-Boot-Framework-Klasse `ResourceSupport`, wodurch die WebAPI-Responses gemäß der JSON-HAL-Spezifikation zusätzliche Daten beinhalten können.
  - Beispiel: Ein `CustomerResource`-Objekt kapselt ein `Customer`-Objekt. Im Constructor von `CustomerResource` wird zusätzlich dafür gesorgt, dass sich in WebAPI-Responses nach Aufruf von `CustomerController.getCustomer(id)` zusätzlich ein Link (URL) zum Abruf der zugehörigen Shipments befindet.
- Package `services`:
  - Klassen in diesem Paket verarbeiten Aufrufe fachlich, wobei Sie andere Services in Package `services` aufrufen oder Anfragen an die Datenbank erzeugen durch Verwendung von Klassen aus Package `repositories`. Jeder Service besteht dabei aus einem Java-Interface und einer implementierenden Klasse.
  - Beispiel: `ShipmentServiceImpl.createShipment(id)`
    - delegiert die Adressprüfung an `AddressValidationService.getValidationErrors(Address)`
    - delegiert die Label-Erzeugung an `LabelService.createLabel(Address, id)`
    - und speichert bei Erfolg die Sendung mittels `ShipmentRepository.save(Shipment)`
- Package `exceptions`:
  - Bei bestimmten Fehlersitutationen werden in Methoden von Klassen innerhalb von Package `services` Exceptions erzeugt, die im Package `exceptions` implementiert sind.
  - Beispiel: `ShipmentServiceImpl.createShipment(id)` erzeugt eine `InvalidAddressException`, wenn ein Shipment (Versandauftrag) auf Basis einer ungültigen Empfängeradresse erzeugt werden soll.
- Package `repositories`:
  - Das Package `repositories` enthält Interfaces, die von der Spring-Boot-Framework-Klasse `CrudRepository` erben. Das Spring-Boot-Framework erzeugt hieraus automatisch zur Laufzeit implementierende Klassen und Objekte, über die einfache und übliche Datenbankzugriffe realisiert werden.
  - Beispiel: `ShipmentServiceImpl.createShipment(id)` ruft `ShipmentRepository.save(Shipment)` auf. Die

Implementierung geschieht durch Spring-Boot automatisch, d.h. es gibt keine `ShipmentRepositoryImpl.save(Shipment)`.

Die Klassen, die die vier Anwendungsfälle implementieren, sind gemäß ihrer technischen Rolle auf die oben dargestellten Packages verteilt. Die Package-Struktur unterteilt die Klassen also nicht bezüglich ihrer fachlichen Zugehörigkeit zu Anwendungsfällen. Beispielsweise befinden sich in Package `services` die Klassen `ShipmentServiceImpl` und `CustomerServiceImpl`, die zur Implementierung verschiedener Anwendungsfälle dienen.

### Verhalten

In [Verarbeitung eines createShipment-WebAPI-Requests] ist die Verarbeitung eines WebAPI-Requests als UML-Sequenzdiagramm dargestellt am Beispiel eines createShipment-Requests. Aus jedem Package ist hierbei mindestens eines Klasse beteiligt.

*Verarbeitung eines createShipment-WebAPI-Requests*

C:\work\sm\parcer\createShipment.png

### Maven-Modul-Struktur

Parcer besteht aus einem einzigen Maven-Modul, d.h. der Build wird auch nur durch eine ein `pom.xml` beschrieben und erzeugt genau ein Artefakt `parcer.jar`.

Die `pom.xml` deklariert mehrere Abhängigkeiten zu 3rd-Party-Libraries, insbesondere:

- Spring-Boot: Parcer nutzt das im Java-Umfeld beliebte Spring-Boot-Framework. Spring-Boot befreit Entwickler von bestimmten Implementierungsaufgaben, wie die (De-)Serialisierung von WebAPI-Request/Responses oder die Implementierung einfacher Datenbankzugriffe.
- Thymeleaf: Der Erzeugung eines Labels als PDF geschieht unter Zuhilfenahme der Bibliothek Thymeleaf.

### Git-Repository-Struktur

Für die Entwicklung von Parcer gibt es ein führendes Git-Repository. Dort ist sämtlicher Quellcode von Parcer zu finden.

### Datenbank

Parcer verwendet eine H2-Datenbank. Annotationen wie `@Entity` der Java-Persistence-API in den Klassen des `domain`-Packages sorgen dafür, dass das Datenbank-Schema automatisch aufgebaut wird. Separate DDL-SQL-Dateien sind hier nicht notwendig. (Aus Gründen der Einfachheit, wird hier die Datenbank nur "in memory" gehalten.)

Abbildung 4 fasst die monolithische Struktur aus Deployment-Sicht zusammen.

*Monolithische Struktur als UML-Deployment-Diagramm*

image

## Parcer herunterladen

Führen Sie in einer Bash folgenden Befehl aus, der Parcer in der Ausgangsstellung herunterlädt und im aktuellen Verzeichnis ~/sm/parcer entpackt. (Das Verzeichnis können Sie natürlich selbst wählen.)

```
mkdir -p ~/sm/parcer
curl -skL https://bit.ly/sm1819-parcer | tar xvfz - -C ~/sm/parcer
```

Wenn Sie ein JDK 11 oder höher einsetzen (mittels java -version kontrollierbar), fügen Sie bitte in ~/sm/parcer/pom.xml noch folgende <dependency> ein

```
 dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
```

## Parcer unter Versionskontrolle stellen

Sorgen Sie dafür, dass das `parcer`-Verzeichnis unter (zunächst lokale) Git-Versionskontrolle gestellt wird.

**IMPORTANT** | Wie bewerkstelligen Sie das? --> git init

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=under_git]

Erzeugen Sie anschließend für Ihr Team ein eigenes Remote-Repository, in dem Sie die Weiterentwicklungen von Parcer versionieren. Setzen Sie die Remote-Ref origin auf Ihr Remote-Repository.

**IMPORTANT** | Wie bewerkstelligen Sie das?

```
- git remote add origin git@gitlab.com:leonelnguimatsia1997/ parcet.git
- git add .
- git push -u origin master
```

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=add_origin]

Rufen Sie Maven derart auf, dass `parcer-0.0.1-SNAPSHOT.jar` erzeugt und im lokalen Repository installiert wird

**IMPORTANT** | Wie bewerkstelligen Sie das? mvn clean package -U

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=maven_install]

## Parcer starten

Starten Sie Parcer lokal mittels

```
mvn spring-boot:run
```

**CAUTION** | Dabei wird vorausgesetzt, dass Port `8080` noch durch keinen Prozess belegt ist.

## Requests absetzen

Parcer stellt keine WebGUI bereit, sondern nur eine http-basierte, "RESTful" API, die JSON+HAL-Repräsentationen in Requests und Responses erwartet bzw. produziert.

Folgender Request erzeugt einen neuen Kunden in der Parcer-Datenbank

```
POST /customers HTTP/1.1
Host: localhost:8080

{"name":"TH Koeln"}
```

Effektiv wird hier nur ein Key-Value-Paar übergeben.

Erzeugen Sie einen solchen Request, beispielsweise unter Zuhilfenahme des Kommandozeilen-Tools `curl`, das auch unter Windows im Zuge der Installation von Git in der GitBash verfügbar sein sollte.

```
curl -sSL -D - -H "Content-Type: application/json" -X POST \
-d '{"name":"TH Koeln"}' http://localhost:8080/customers
```

Als Response wird retourniert:

```
{
    "customer": {
        "id": 1,
        "name": "TH Koeln"
    },
    "_links": {
        "shipments": {
            "href": "http://localhost:8080/customers/1/shipments"
        },
        "self": {
            "href": "http://localhost:8080/customers/1"
        }
    }
}
```

Die übergebenen Daten werden angereichert über eine von Parcer vergebene id für den Kunden. Der Response beinhaltet konform zur JSON+HAL-Spezifikation darüber hinaus Links zu assoziierten Daten, nämlich einen Link zu dem Kunden zugeordnete Shipments (die im Moment noch nicht vorhanden sind) und einem `self`-Link, über den die erzeugte Customer-Ressource später erneut abgerufen werden kann.

In den JavaDocs der Methoden der Klassen innerhalb des Packages `controllers` finden Sie weitere `curl`-basierte Beispielaufrufe.

Nehmen Sie diese als Beispiele, um folgende Testfälle manuell durchzuführen:

1. Anlegen eines Kunden (ist oben bereits geschehen)
2. Validierung einer postalischen Adresse (beispielsweise Ihrer Privat-Adresse) Valiedierte Adresse ohne Fehlermeldung: 51503
3. Validierung einer postalischen Adresse mit einer fehlenden Ziffer in der Postleitzahl {"_embedded":{"validationErrorResourceList":[{"validationError":"nogermanzipcodefound"}]}}-->Fehlermeldung
4. Erzeugung eines PDF-Labels mit beliebiger, valider recipientAddress und beliebiger shipmentNumber Ohne Fehlermeldung
5. Erzeugung einer Sendung mit beliebiger, valider recipientAddress - curl -s http://localhost:8080/customers -curl-sSL-D--H"Content-Type:application/json" -XPOST-d'{"recipientAddress":{"line1":"ReneWoerzberger","line2":"Claudiusstrasse16","line3":"51503Roesrath"}}http://localhost:8080/customers/1/shipments
6. Abruf des in Schritt 5 erzeugten Label - curl -s http://localhost:8080/labels/2
7. Abruf aller Sendungen des in Schritt 1 erzeugten Kunden.

```
curl -s http://localhost:8080/customers/1/shipments
```

**IMPORTANT** | Notieren Sie sich die Aufrufe, die Sie getätigt haben.

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=sol-parcertestcurls]

## H2-Console verwenden

Öffnen Sie im Browser http://localhost:8080/h2-console und loggen Sie sich mit

- der JDBC URL `jdbc:h2:mem:testdb`
- User `sa` und
- leerem Passwort ein.

**IMPORTANT** | Wie können Sie den Inhalt der Tabelle `SHIPMENT` ausgeben? SELECT * FROM SHIPMENT

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=sol-selectshipment]

## Messung der Code-Abdeckung

Die Automatisierung der Qualitätssicherung ist in Parcer nur spärlich ausgebaut. In

`parcer\src\test\java\de\thk\parcer\parcer\services\ShipmentServiceImplTest.java` sind lediglich zwei JUnit-Testfälle implementiert.

Messen Sie im Maven-Build die durch die vorhandenen Testfälle erreichte Code-Abdeckung Verwenden Sie hierfür das Maven-JaCoCo-Plugin, das im Productmodel-Beispiel in der Vorlesung/Übung bereits eingesetzt wurde.

## HTML-Report

Stellen Sie sicher, dass während des Maven-Builds in `target/site/jacoco` ein HTML-Report zur erreichten Code-Abdeckung generiert wird.

## JaCoCo-Dokumentation

Informieren Sie sich mittels

```
mvn help:describe -Dplugin=org.jacoco:jacoco-maven-plugin -Ddetail
```

über weitere Goals von JaCoCo.

## Mindestabdeckung

| | |
|---|---|
| **IMPORTANT** | Stellen Sie durch Einbindung des passenden JaCoCo-Goals sicher, dass der Maven-Build abbricht, sobald die Anweisungsüberdeckung im Package `de.thk.parcer.parcer.services` weniger als `85%` beträgt. Verwenden Sie das Goal `check`, das in der Phase `prepare-package` ausgeführt werden soll. |

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=sol-mindestabdeckungmaven]

## Erweiterung der Code-Abdeckung

Die vorhandenen JUnit-Testfälle führen zu einer Anweisungsüberdeckung von `76%` im Package `de.thk.parcer.parcer.services`. Der Maven-Build bricht daher ab, sofern die Aufgabe Mindestabdeckung korrekt gelöst wurde.

| | |
|---|---|
| **IMPORTANT** | Analysieren Sie mittels des generierten Reports in `target/site/jacoco`, welche Anweisungen in Package `de.thk.parcer.parcer.services` nicht überdeckt sind. Erzeugen Sie eine Klasse `parcer\src\test\java\de\thk\parcer\parcer\services\LabelServiceImplTest.java`, mit der Sie die Abdeckung in `de.thk.parcer.parcer.services.LabelService` erhöhen, so dass Sie in dem Package eine Anweisungsabdeckung von mindestens `85%` erreichen. Orientieren Sie sich dabei an `parcer\src\test\java\de\thk\parcer\parcer\services\ShipmentServiceImplTest.java`. |

Unresolved directive in <stdin> - include::lab_quality_assurance_solutions_de.asciidoc[tag=sol-erweiterungabdeckung]

## JMeter

Starten Sie JMeter. Sie können die JMeter-GUI ohne Installation als Maven-Plugin beispielsweise direkt von der Kommandozeile starten, ohne ein <plugin> in der pom.xml konfigurieren zu müssen:

```
mvn com.lazerycode.jmeter:jmeter-maven-plugin:2.8.0:gui
```

Erstellen Sie einen Testplan, den Sie unter src/test/resources/parcer.jmx speichern.

Der Testplan soll einen Customer anlegen und in 100 nebenläufigen Threads jeweils 10 (also insgesamt 1000) Shipments.

Hinweise:

- Maven wird in einer Version 3.5.0 oder höher benötigt.
- Sie müssen bei allen http-Requests den http-Header `Content-Type: application/json` setzen.
- Über die `Config Element`s `HTTP Header Manager` und `HTTP Requests Defaults` können Sie Gemeinsamkeiten der http-Requests an einer Stelle konfigurieren.
- Sie können bei der Anlage der Shipments die ID des angelegten Customers hartkodieren. Sie sollte bei Anlage nur eines Customers `1` sein.
- Vergessen Sie bitte nicht, die `parcer.jmx` unter Versionskontrolle zu stellen.

## Sonarqube (optionale Aufgabe)

Wenn Sie Docker lokal installiert haben, starten Sie einen Sonarqube-Container mittels

```
docker container run -d --name sonarqube -p 9000:9000 -p 9092:9092 sonarqube
```

Überprüfen Sie, ob die Instanz wirklich läuft, indem Sie http://localhost:9000 im Browser öffnen.

Analysieren Sie das Projekt mittels

```
mvn org.sonarsource.scanner.maven:sonar-maven-plugin:3.5.0.1254:sonar
```

und refreshen Sie die Seite im Browser.

Welche Mängel wurden gefunden?

Stoppen Sie anschließend den Sonarqube-Docker-Container mittels

```
docker container stop sonarqube
```

## Zerlegung in Maven-Module (optionale Aufgabe)

Zerlegen Sie das Maven-Modul `parcer`. Das Modul `parcer` soll nur noch als Aggregator-Modul verbleiben und die Module `addressvalidation`, `label`, `shipment`, `customer` und `core` aggregieren. Der unverändert bleibende Quellcode (`*.java`) soll wie folgt aufgeteilt werden:

- Modul `core` erhält die Packages `domain` und `exceptions` und sämtliche bislang darin enthaltende Klassen.
- Module `label`, `addressvalidation`, `customer` und `shipments` haben die Packages `controllers`, `services`, `resources` und `repositories`. Darin enthalten sind die jeweils jedoch nur die für den Anwendungsfall relevanten Klassen. Beispielsweise ist im Modul `customer` im Package `controllers` nur die Klasse `CustomerController`.
- Die Dependencies zwischen den Modulen soll in der jeweiligen `pom.xml` so konfiguriert werden, wie in Abbildung 6 dargestellt.

*Maven-Modulstruktur mit Abhängigkeiten*

image

Stellen Sie sicher, dass nach der Zerlegung Parcer nach wie vor mit

```
mvn install
mvn spring-boot:run
```

gebaut und gestartet werden kann.

Hinweise:

- Die Aufgabe besteht im Wesentlich im Anlegen von Verzeichnissen, Anlegen oder Änderung von `pom.xml`s und im Verschieben von `*.java`-Dateien, die nur angepasst werden sollen, wenn Abhängigkeiten unbedingt aufgelöst werden müssen.
- Sie können die aggregierende `parcer/pom.xml` auch als Parent-POM verwenden. Überlegen Sie sich gut, welche `<dependencies>` in welchem Modul überhaupt gebraucht werden. (Wenn eine fehlt, merken Sie das spätestens beim `mvn install` oder ggf. früher in der Entwicklungsumgebung: IntelliJ IDEA markiert fehlende Importe (nach Änderungen in pom.xmls + Klick auf Import Changes) als Fehler). Nutzen Sie auch das `<dependencyManagement>` in der Parent-POM.
- Leider ist das Zusammenspiel von Spring-Boot, Unit-Tests und Maven-Submodulen nicht ganz einfach. Auch wenn es architektonisch unsauber ist, verschieben Sie sämtliche Testfälle (unterhalb von `src/test`) in `shipment/src/test` und die `ParcerApplication.java` auch an die entsprechende Stelle unterhalb von `shipement/src`.

Last updated 2022-12-19 12:57:14 +0100