

# I\_BA\_SWE\_FS2018 Dokumentation SWE

P. Inäbnit & D. Schafer

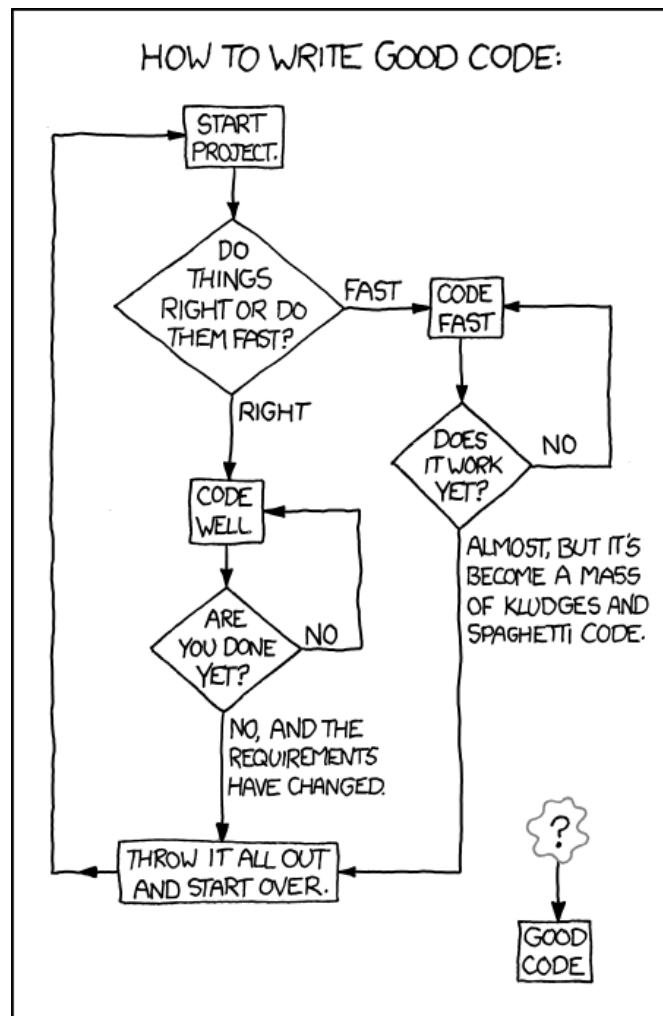


Abbildung 1: <https://xkcd.com/844/>

May 29, 2018, Luzern

# 1 Introduction

## 1.1 About

This documentation has been created during our studies at the **Lucerne University of Applied Sciences and Arts, Information Technology**. Despite of carefully checking this documentation, there is no guarantee for completion, correctness and accuracy of the covered content.

Most of the information comes from presentations of our honored teachers, Mr. Gisler and Mr Šučur.

## 1.2 Contribution

If you would like to contribute to this documentation or to report any incorrection, please contact one of the authors of this document:

Pascal Inäbnit: `pascal.inaebnit@stud.hslu.ch`

David Schafer: `david.schafer@stud.hslu.ch`

## 1.3 Licence

This work is licensed under a creative commons license. (Attribution-NonCommercial-ShareAlike 4.0 International)



You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. NonCommercial — You may not use the material for commercial purposes. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. <https://creativecommons.org/licenses/by-nc-sa/4.0/>

# Inhaltsverzeichnis

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	About . . . . .	1
1.2	Contribution . . . . .	1
1.3	Licence . . . . .	1
<b>2</b>	<b>Abstract</b>	<b>3</b>
2.1	Aufgabenstellung . . . . .	3
2.2	Die Lösung der Autoren . . . . .	3
2.3	Phase 2 . . . . .	3
<b>3</b>	<b>Architektur</b>	<b>4</b>
3.1	REST Schnittstelle Möbelhaus . . . . .	5
3.2	REST Collector . . . . .	6
3.3	MongoDB . . . . .	6
3.4	MVA Core System . . . . .	7
3.5	Clients . . . . .	7
<b>4</b>	<b>Module/Komponenten</b>	<b>8</b>
4.1	Collector-Modul . . . . .	8
4.2	CORE-Modul . . . . .	8
4.3	Client-Modul . . . . .	8
<b>5</b>	<b>Klassendiagramme</b>	<b>9</b>
<b>6</b>	<b>Implementation</b>	<b>13</b>
6.1	Datenbank . . . . .	13
6.2	Core Server . . . . .	14
6.3	Datacollector . . . . .	15
6.4	Client . . . . .	17
<b>7</b>	<b>Lessons Learned</b>	<b>21</b>
7.1	Übersicht schaffen . . . . .	21
7.2	Herausforderungen . . . . .	21
7.2.1	JSON, GSON, Jackson, was ist das eigentlich? . . . . .	21
7.2.2	Architektur Entscheid - Datenbank . . . . .	22
7.2.3	Erarbeitung der funktionalen Anforderungen . . . . .	22
7.2.4	Daily struggle with Jenkins . . . . .	26
7.3	RMI Testing . . . . .	26
7.4	Kohäsion und Kopplung . . . . .	26
7.5	Fazit . . . . .	27

## 2 Abstract

### 2.1 Aufgabenstellung

Für den Möbelherstellerverband (MHV) soll eine Applikation erstellt werden, mit der verschiedene Auswertungen zur Analyse der Verkäufe durchgeführt werden können. Dazu stellen alle Möbelhersteller (Mitglieder des MHW) eine auf REST basierende Schnittstelle zur Verfügung, mit der Daten im JSONFormat abgefragt werden können.

Jeder Möbelhersteller verwaltet seine eigenen Produkte (Möbel), nimmt Bestellungen von unterschiedlichen Möbelhäusern an, und liefert das bestellte Möbel an die Möbelhäuser aus.

Über die Schnittstelle lassen sich Informationen zu den im Angebot stehenden Produkten, den Möbelhäusern, den Bestellungen und den Lieferungen abfragen.

### 2.2 Die Lösung der Autoren

Die Lösung der Autoren basiert auf einer verteilten Architektur, welche im Kapitel 'Architektur' näher beschrieben wird.

### 2.3 Phase 2

Die in der Phase 2 erledigten Arbeiten waren:

- Vollständige Client Integration **ok**
- GUI Client **ok**
- Anfrageoptimierung der Anforderungen A01 bis A09 **ok**
- Eigenständige Java Container für die Module **ok**

### 3 Architektur

Die Architektur der Lösung basiert auf dem Wunsch der Autoren, einzelne Komponenten in einer Container-Basierten Umgebung zu betreiben.

Daher werden sowohl die Datenbank (MongoDB<sup>1</sup>) als auch der CoreServer und der Collector, der entwickelten Lösung, in Docker-Containern auf einem Host im Enterpriselab<sup>2</sup> betrieben.

Die Verwaltung dieser Umgebung geschieht mit der Software Portainer<sup>3</sup>, welche ebenfalls in einem Container läuft.

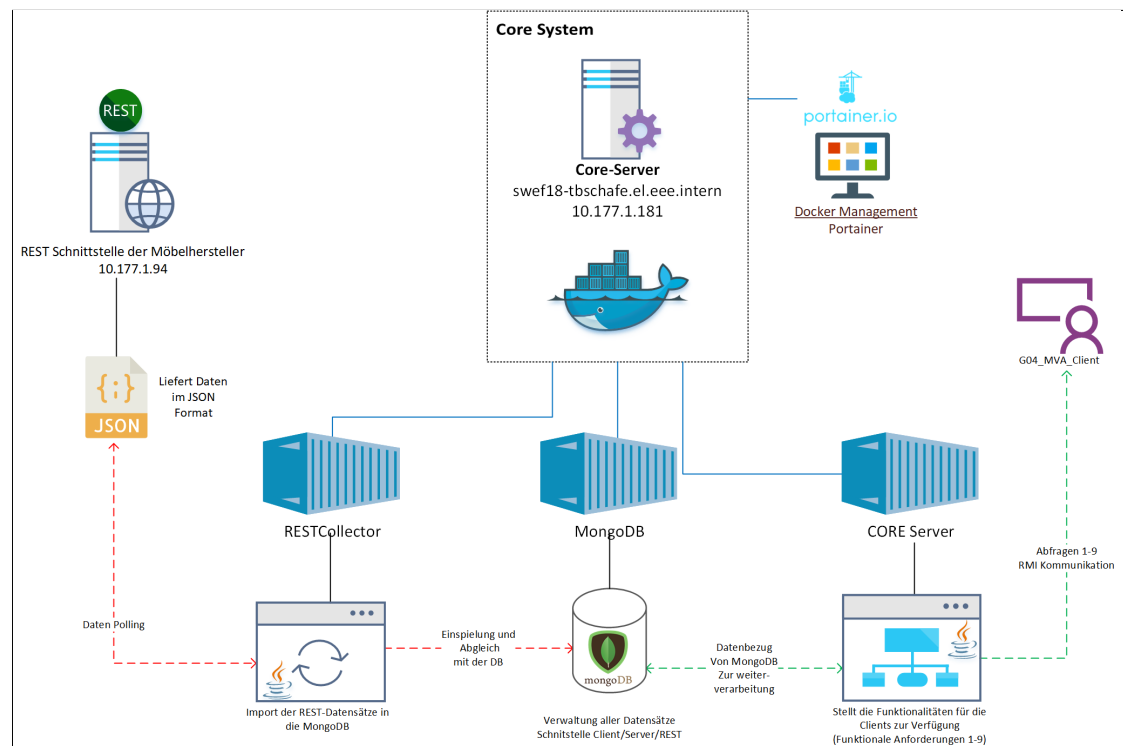


Abbildung 2: Systemarchitektur G04

Grundsätzlich wollen die Autoren mit dieser Lösung eine hohe Modularität und Unabhängigkeit (sprich eine lose Kopplung) erreichen, weshalb für jeden Aufgabenbereich ein eigenes Service Modul verwendet wird.

Zum Beginn der Arbeit wurde entschieden, dass alle Daten von der REST Schnittstelle in eine Datenbank übertragen und konsequent von da aus weiterverarbeitet werden.

Die Überlegung dahinter ist, dass in der Praxis oftmals kleine Webserver-Appliances bei den Kunden bereitgestellt werden, welche die nötigsten Daten aufbereiten. Diese sind jedoch nicht für viele Anfragen ausgelegt und stossen rasch an ihre Kapazitätsgrenzen.

Ausserdem kann man sich gut vorstellen, dass in der Praxis die REST Server an verschiedenen Standorten, jeweils hinter einer Firewall stehen und nicht unbedingt von überall

<sup>1</sup><https://www.mongodb.com/>

<sup>2</sup><https://www.enterpriselab.ch/>

<sup>3</sup><https://portainer.io/>

öffentlich zugänglich sind.

### 3.1 REST Schnittstelle Möbelhaus

Die Schnittstelle der Möbelhersteller wird von der Hochschule zur Verfügung gestellt und stellt eine Art Blackbox dar. Es werden drei Möbelhäuser und eine Test Instanz zur Verfügung gestellt, welche über folgende Basis URLs erreichbar sind:

Nr.	Name	Web Service URL
1	Holzmöbel Fischer AG Bergstrasse 28 6440 Brunnen	<a href="http://10.177.1.94:8081/rmhr-fischer/">http://10.177.1.94:8081/rmhr-fischer/</a>
2	Möbelfabrik Walker AG Bundesstrasse 44 6280 Hochdorf	<a href="http://10.177.1.94:8082/rmhr-walker/">http://10.177.1.94:8082/rmhr-walker/</a>
3	Möbelfabrik Zwissig GmbH Zwyergasse 17 6460 Altdorf	<a href="http://10.177.1.94:8083/rmhr-zwissig/">http://10.177.1.94:8083/rmhr-zwissig/</a>

*Tabelle 1 – Möbelhersteller Endpoints (URIs)*

Zusätzlich existiert eine Test-Instanz die während der Entwicklung für Integrationstests verwendet werden kann. Diese liefert im Gegensatz zu den drei Live-Systemen konstante, unveränderte Daten.

Nr.	Name	Web Service URL
1	Test-Instanz Liefert konstante Daten.	<a href="http://10.177.1.94:8090/rmhr-test/">http://10.177.1.94:8090/rmhr-test/</a>

Abbildung 3: Basis URLs der verschiedenen Webservices

Pro Schnittstelle stehen zehn grundsätzliche Befehle zur Abfrage sowie Filterung der Daten zur Verfügung:

Nr.	URL	Beschreibung	Beispiel
1	host_ip:port/rmhr-test/ws/moebelhaus	Liefert alle Möbelhäuser zurück.	<a href="http://10.177.1.94:8090/rmhr-test/ws/moebelhaus">http://10.177.1.94:8090/rmhr-test/ws/moebelhaus</a>
2	host_ip:port/rmhr-test/ws/moebelhaus/code/MOEBELHAUS_CODE	Liefert das Möbelhaus zurück, dessen Code übergeben wird.	<a href="http://10.177.1.94:8090/rmhr-test/ws/moebelhaus/code/MH_DIGA_EMME">http://10.177.1.94:8090/rmhr-test/ws/moebelhaus/code/MH_DIGA_EMME</a>
3	host_ip:port/rmhr-test/ws/moebelhaus/name?value=MOEBELHAUS_NAME	Liefert das Möbelhaus für den übergebenen Namen zurück.	<a href="http://10.177.1.94:8090/rmhr-test/ws/moebelhaus/name?value=diga_möbel_ag">http://10.177.1.94:8090/rmhr-test/ws/moebelhaus/name?value=diga_möbel_ag</a>
4	host_ip:port/rmhr-test/ws/katalog	Liefert alle Produkt-Typen zurück.	<a href="http://10.177.1.94:8090/rmhr-test/ws/katalog">http://10.177.1.94:8090/rmhr-test/ws/katalog</a>
5	host_ip:port/rmhr-test/ws/katalog/typ/code/TYP_CODE	Liefert den Produkt-Typ zurück, dessen (eindeutiger) ProduktTyp-Code übergeben wird.	<a href="http://10.177.1.94:8090/rmhr-test/ws/katalog/typ/code/PCTI-Steve-2011">http://10.177.1.94:8090/rmhr-test/ws/katalog/typ/code/PCTI-Steve-2011</a>
6	host_ip:port/rmhr-test/ws/katalog/typ/name?value=TYP_NAME	Liefert alle Produkt-Typen für den übergebenen ProduktTyp-Name zurück.	<a href="http://10.177.1.94:8090/rmhr-test/ws/katalog/typ/name?value=Schrack">http://10.177.1.94:8090/rmhr-test/ws/katalog/typ/name?value=Schrack</a>
7	host_ip:port/rmhr-test/ws/katalog/lagerbestand?code=TYP_CODE	Liefert den Lagerbestand des Produkt-Typs zurück, dessen Produkt-Typ Code übergeben wird.	<a href="http://10.177.1.94:8090/rmhr-test/ws/katalog/lagerbestand/typ/code/DSKT-Mike-8901">http://10.177.1.94:8090/rmhr-test/ws/katalog/lagerbestand/typ/code/DSKT-Mike-8901</a>
8	host_ip:port/rmhr-test/ws/bestellung/moebelhaus?code=MOEBELHAUS_CODE	Liefert die Bestellungen des Möbelhauses zurück, dessen Code übergeben wird.	<a href="http://10.177.1.94:8090/rmhr-test/ws/bestellung/moebelhaus?code=MH_DIGA_EMME">http://10.177.1.94:8090/rmhr-test/ws/bestellung/moebelhaus?code=MH_DIGA_EMME</a>
9	host_ip:port/rmhr-test/ws/bestellung/offen/moebelhaus?code=MOEBELHAUS_CODE	Liefert die offenen Bestellungen des Möbelhauses zurück, dessen Code übergeben wird.	<a href="http://10.177.1.94:8090/rmhr-test/ws/moebelhaus?code=MH_DIGA_EMME">http://10.177.1.94:8090/rmhr-test/ws/moebelhaus?code=MH_DIGA_EMME</a>
10	host_ip:port/rmhr-test/ws/lieferung/moebelhaus?code=MOEBELHAUS_CODE	Liefert alle Lieferungen des Möbelhauses zurück, dessen Code übergeben wird.	<a href="http://10.177.1.94:8090/rmhr-test/ws/lieferung/moebelhaus?code=MH_DIGA_EMME">http://10.177.1.94:8090/rmhr-test/ws/lieferung/moebelhaus?code=MH_DIGA_EMME</a>

Abbildung 4: Funktionen REST Schnittstelle

## 3.2 REST Collector

Der REST Collector stellt das Bindeglied zwischen den Möbelherstellern (REST Schnittstelle) und der zentralen Datenverwaltung (MongoDB) dar.

Folgende Daten werden vom REST Collector in die Datenbank importiert:

- Möbelhäuser (Kunden) pro Möbelhersteller
- Bestellungen pro Möbelhaus
- Produkte des Möbelherstellers
- Lieferungen pro Möbelhaus

## 3.3 MongoDB

Zu Beginn des Projekts wurde MongoDB als zentrale Datenbank ausgewählt. Dies, da die Daten einerseits vom REST Service als JSON daher kommen und MongoDB prädestiniert ist für JSON Daten. Andererseits bietet MongoDB schnelle Datenzugriffe und Verarbeitung und ist somit eine optimale Wahl für Mehrbenutzer Zugriffe.

Folgende Collections stellt die MongoDB zur Verfügung:

- MH\_Möbelhersteller
  - Alle Möbelhäuser welche der Möbelhersteller als Kunde beliefert
- Bestellungen\_MöbelhausCode\_Möbelhersteller
  - Alle Bestellungen pro Möbelhaus des Möbelherstellers
- Produkte\_Möbelhersteller
  - Alle Produkte des Möbelherstellers
- Lieferungen\_MöbelhausCode\_Möbelhersteller
  - Alle Lieferungen pro Möbelhaus des Möbelherstellers

### 3.4 MVA Core System

Das MVA Core System ist der zentrale Server, an welchen sich die Clients wenden. Er bezieht seine Daten von der Datenbank und verarbeitet diese entsprechend, um die Ergebnisse den Clients zur Verfügung zu stellen.

Folgende neun Funktionen stellt der Server den Clients zur Verfügung (*Auszug aus dem Interface "FurnitureManufacturerInterface"*):

- getMoebelhauser01();
- getProductTypes02();
- getAverageOrderValuePerFurnitureShop03();
- getOrderValueFromPeriod04(Date Von, Date Bis);
- getTop3FurnitureShops05(Date Von, Date Bis);
- getAverageDeliveryTime06();
- getTop5Products07(Date Von, Date Bis);
- getOrdersForAllWeeks08();
- getOrderVolumeForAllWeeks09();

Diese Funktionen korrespondieren entsprechend mit den funktionalen Anforderungen des Projekts:

ID	Abfrage
A01	Wieviele Möbelhäuser sind bei einem Möbelhersteller als Kunde registriert?
A02	Wieviele verschiedene Produkt-Typen hat ein Möbelhersteller in seinem Angebot?
A03	Wie gross ist der durchschnittliche Wert einer Bestellung pro Möbelhaus bei einem Möbelhersteller?
A04	Wie gross ist der Bestellungswert pro Möbelhersteller in einer bestimmten Zeitperiode?
A05	Welche drei Möbelhäuser haben das grösste Bestellschlagvolumen (Wert) bei einem Möbelhersteller? (Optional: in einer vorgegebenen Zeitperiode)?
A06	Wie lange ist die durchschnittliche Lieferzeit eines Möbelherstellers?
A07	Welche fünf Produkte wurden bei einem Möbelhersteller am häufigsten verkauft? (Optional: in einer vorgegebenen Zeitperiode)
A08	Anzahl Bestellungen pro Möbelhaus und Kalenderwoche des aktuellen Kalenderjahres? <sup>1</sup>
A09	Bestellschlagvolumen pro Möbelhaus und Kalenderwoche des aktuellen Kalenderjahres?

Abbildung 5: Funktionale Anforderungen

### 3.5 Clients

Der Client kann auf jedem System in Betrieb genommen werden, auf welchem Java installiert ist und von welchem eine Netzwerkverbindung zum EnterpriseLab aufgebaut werden kann (für diese Testinstallation). Der Client braucht lediglich Zugriff auf den MVA Core Server, über welchen die Daten bereit gestellt werden. Die Daten werden via RMI<sup>4</sup> Schnittstelle bezogen.

<sup>4</sup><https://docs.oracle.com/javase/tutorial/rmi/index.html>



## 4 Module/Komponenten

In diesem Kapitel wird die Aufteilung der verschiedenen Module erklärt. Zudem wird gezeigt, welche Funktionen diese enthalten. Die Autoren haben sich für die Aufteilung in drei separate Java Kernmodule entschieden, welche unabhängig voneinander agieren können.

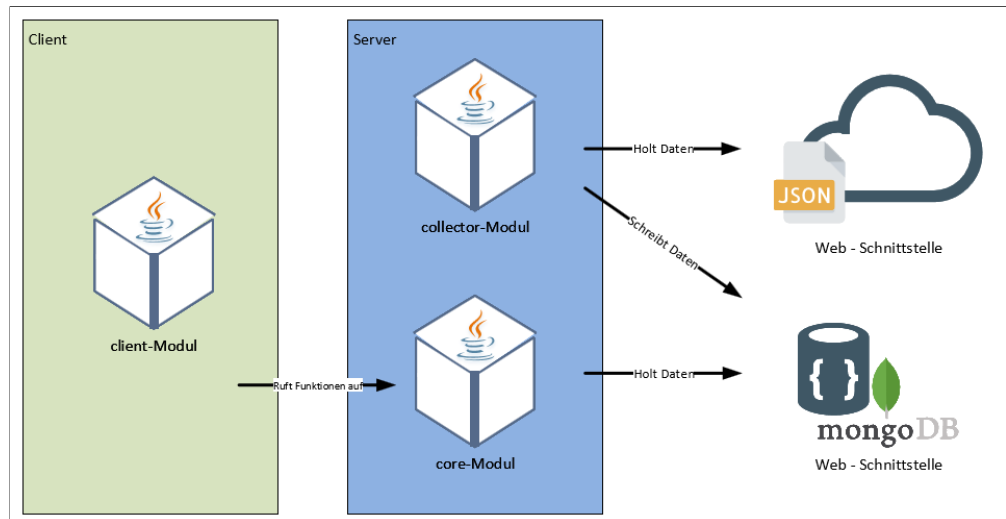


Abbildung 6: Module/Komponenten

### 4.1 Collector-Modul

Der RESTCollector ist zuständig für das Sammeln der Daten, die von der jeweiligen REST<sup>5</sup> Schnittstelle des Möbelherstellers angeboten werden. Die Daten werden aggregiert und in der Datenbank persistiert.

### 4.2 CORE-Modul

Der CORE Server stellt die Verarbeitungslogik den Clients zur Verfügung. Er erfüllt die funktionalen Anforderungen A01 bis A09.

### 4.3 Client-Modul

Der Client stellt die Schnittstelle zum CORE Server zur Verfügung und bietet via RMI die Möglichkeit Daten abzuholen.

<sup>5</sup>[https://de.wikipedia.org/wiki/Representational\\_State\\_Transfer](https://de.wikipedia.org/wiki/Representational_State_Transfer)

## 5 Klassendiagramme

In diesem Kapitel werden die Klassendiagramme, der jeweiligen Module und Komponenten, dargestellt.



Abbildung 7: Klasse mit Schnittstelle, welche die Anforderungen umsetzen

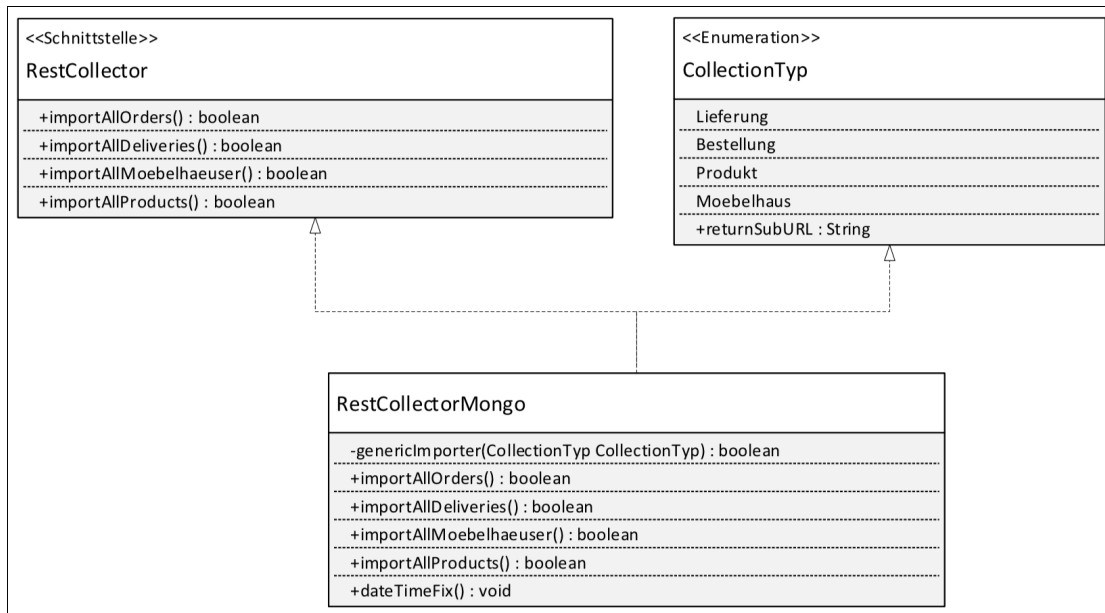


Abbildung 8: Klassen für die Anbindung der REST-Schnittstelle

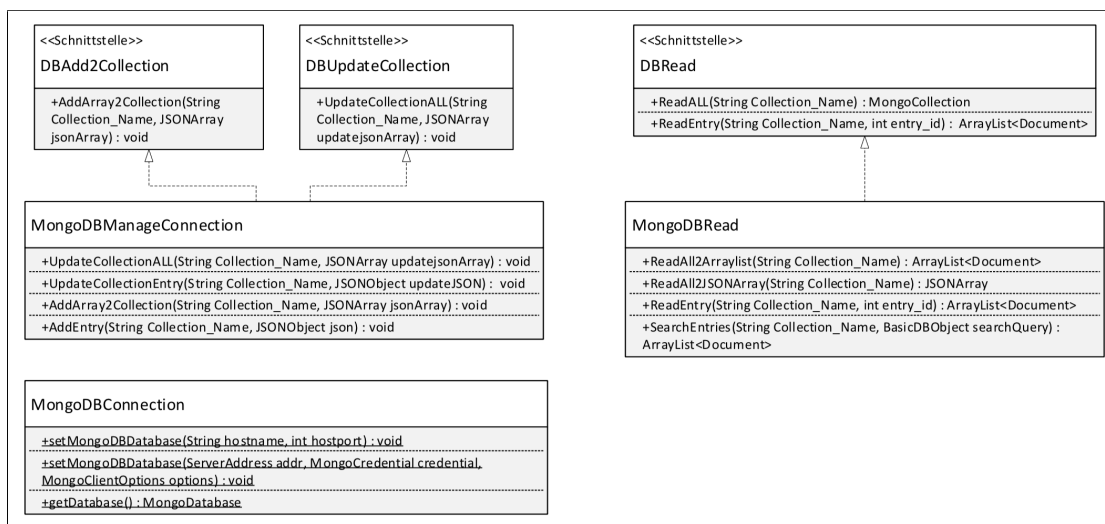


Abbildung 9: Klassen für das DB-Management

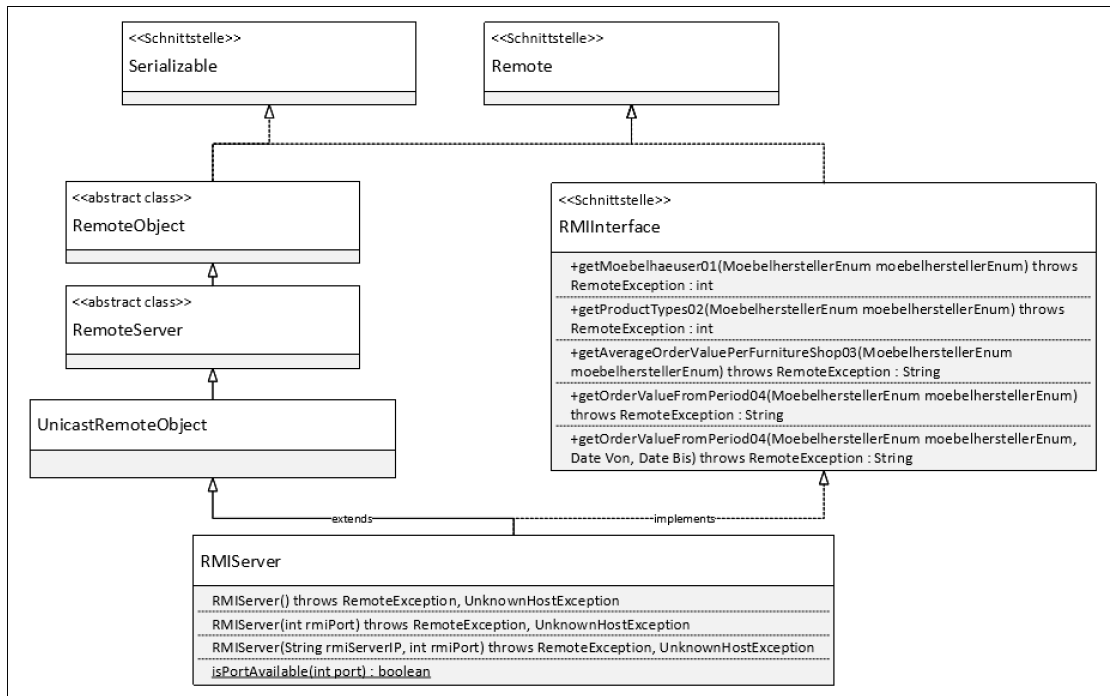


Abbildung 10: Klassen und Schnittstellen für die Implementierung des RMI-Servers

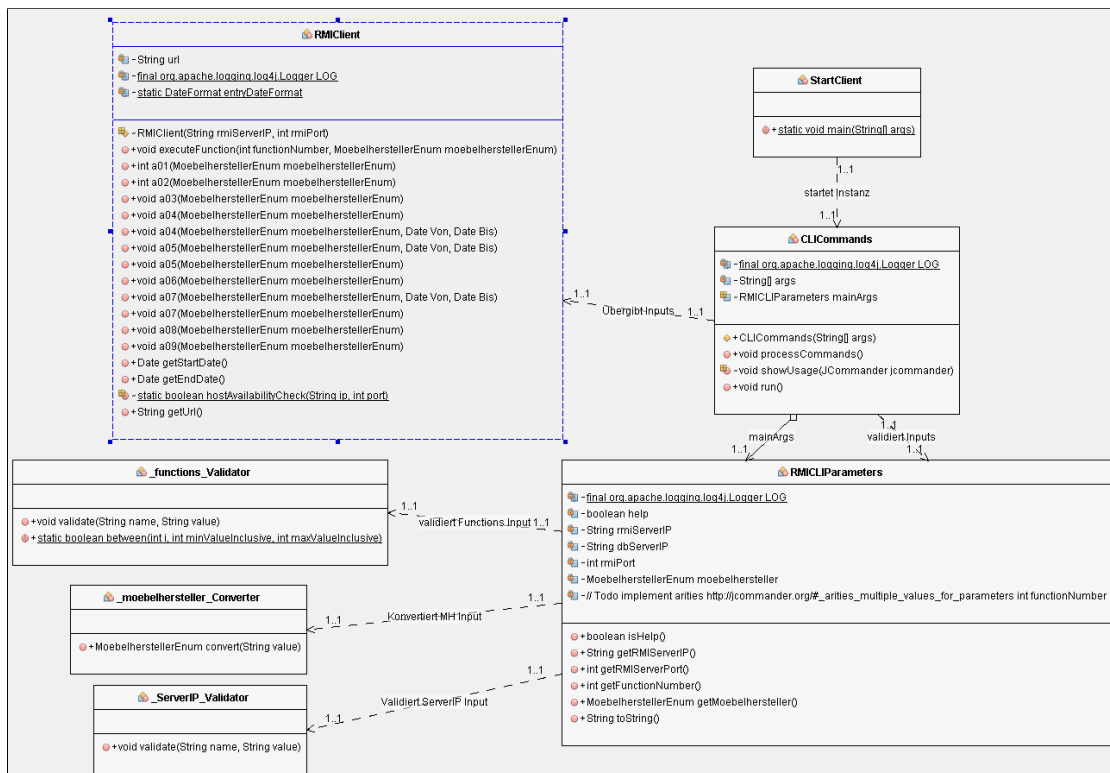


Abbildung 11: Client v2: Via JCommander werden in der Klasse 'RMICLIParameters' Inputs geparkt und validiert, und schlussendlich via 'RMIClient' ausgeführt

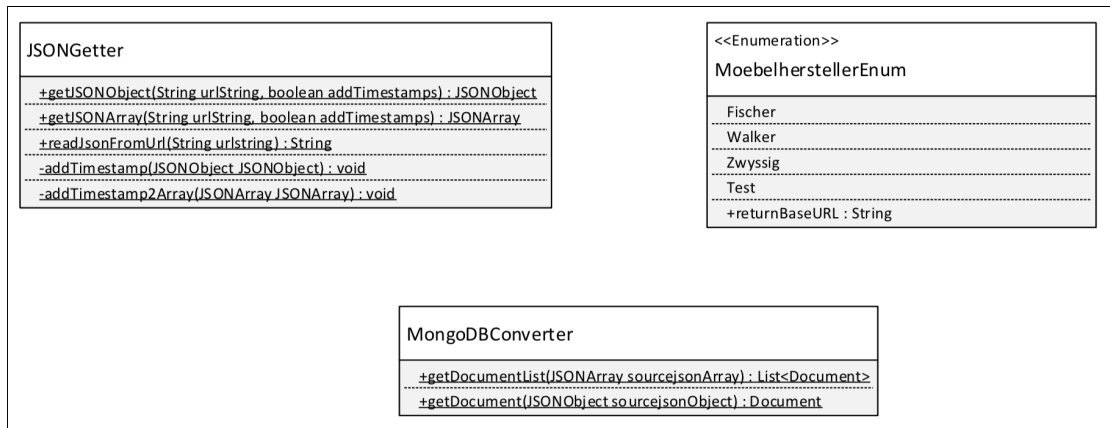


Abbildung 12: Helper - Klassen

## 6 Implementation

In diesem Kapitel wird aufgezeigt, wie die konkrete Implementation der Software aussieht. Dabei wird auf die Module Datenbank, CORE-Server, Datacollector und Client eingegangen.

### 6.1 Datenbank

Das Datenbank Modul ist als MongoDB Container implementiert. Der Container wird von den Entwicklern im offiziellen Docker Hub inklusive Dokumentation angeboten:

[https://hub.docker.com/\\_/mongo/](https://hub.docker.com/_/mongo/)

Nach der Installation des Containers muss man sich via MongoShell oder anderen Tools (z.B. Robo3T<sup>6</sup>) verbinden und eine neue Datenbank instanzieren mittels:

USE MVA\_04

Nun müssen lediglich bei der Initialisierung des CORE-Servers die Parameter 'DB-Server' und 'DB-Name' mitgegeben werden.

Listing 1: Datenbank Einbindung

```
...
public class RMIServer {
    public static void main(String[] args) throws UnknownHostException {

        String mongoHostname = "swef18-tbschafe.el.eee.intern";
        int mongoPort = 27017;
        String DBName = "MVA_04";

        MongoDBConnection.setMongoDBDatabase (mongoHostname, mongoPort, DBName);

        ...
    }
}
```

---

<sup>6</sup><https://robomongo.org/>

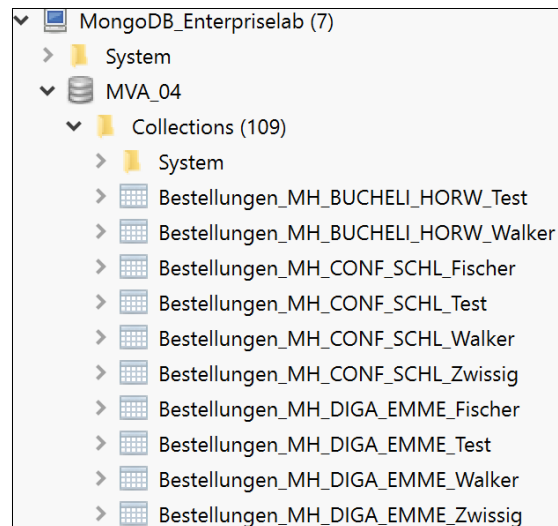


Abbildung 13: DB: Auszug der vom Collector erstellten MongoDB Collections

## 6.2 Core Server

Der Core Server ist als Java+Maven Docker Container implementiert. Der Core Server führt folgende Aufgaben aus:

- RMI Server
- BusinessLogic (Möbelhersteller Abfragen)
- DBConnection (Lesen aus der Datenbank)

Der Container beinhaltet die Komponenten:

- Java 1.8
- Maven 3.1.1
- Git

Der Container wird von James Bloom auf dem Docker Hub bereitgestellt:

<https://hub.docker.com/r/jamesdbloom/docker-java8-maven/>

Eine Continuous Integration und Deployment Konfiguration konnte aus Zeitgründen noch nicht umgesetzt werden, deshalb wird der Container momentan noch manuell konfiguriert:

- Shell in Container aufbauen
- ```
git clone https://gitlab.enteriselab.ch/swe-18fs01/g04-mva.git
```

  - login mit GitLab Account
  - `cd g04-mva`
  - `cd mva-common`
  - `mvn clean compile exec:java`
  - `Dexec.mainClass=ch.hslu.swe.server.RMIServerDocker`

Da RMI eine etwas ältere Technologie ist, müssen bei der Initialisierung in einem Docker Container spezielle Parameter mitgegeben werden :

Listing 2: RMI Server in Docker Umgebung

```
public class RMIServerDockeR extends UnicastRemoteObject
implements RMIIInterface {
    ...
    //Docker Fix 2.1: Die Registry muss statisch sein
    static Registry reg;
    public static void main(String [] args) throws UnknownHostException {
        //docker fix 1: Dieser Parameter muss gesetzt werden da ansonsten
        //die interne Docker IP verwendet wird
        System.setProperty("java.rmi.server.hostname", "10.177.1.181");
        //docker fix 2.2: die Statische Registry wird initialisiert
        reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);

        //Init DB Connection
        MongoDBConnection.setMongoDBDatabase(mongoHostname, mongoPort);

        //Init Interface Objekt
        RMIServerDockeR rmiserverdockeR = new RMIServerDockeR();

        //Rebind: Die registry (statisch) wird auf das neue
        //Objekt (RMIServer) gebunden
        reg.rebind(RMIIInterface.RO_NAME, rmiserverdockeR);

        // Ausfuehrung anhalten (eine Variante)
        LOG.debug("Server gestartet , beenden mit ENTER-Taste!");
        new java.util.Scanner(System.in, "UTF-8").nextLine();

        // Unbind entferntes Objekt
        reg.unbind(RMIServerDockeR.RO_NAME);
        System.exit(0);
        ...
    }
    ...
}
```

## 6.3 Datacollector

Der Datacollector basiert ebenfalls auf der Grundlage des Containers von James Bloom.

Der einzige Unterschied ist der Start der Application via:

```
- mvn clean compile exec:java
-Dexec.mainClass=ch.hslu.swe.Datacollector.StartCollector
```

Der Datacollector holt die Daten asynchron vom REST Server ab, dabei werden zeit-versetzt sogenannte Java TimerTasks<sup>7</sup> gestartet, welche alle notwendigen Imports pro Möbelhersteller durchführen:

- importAllMoebelhauser();
- importAllOrders();

---

<sup>7</sup><https://docs.oracle.com/javase/7/docs/api/java/util/TimerTask.html>



- importAllDeliveries();
- importAllProducts();

Auszug der StartCollector Klasse, welche die Threads startet:

Listing 3: Ausschnitt TimerTask Initialisierung

```
public class StartCollector {

    ...
    public static void main(final String[] args) throws JSONException {

        MongoDBConnection.setMongoDBDatabase(...);

        Calendar startTime_Calendar = Calendar.getInstance();
        long startTimeInMillis = startTime_Calendar.getTimeInMillis();
        Date startTime = new Date(startTimeInMillis);

        LOG.info("RESTCollector Service started at " + startTime);

        //Java TimerThread erstellen und alle 15min pullen lassen
        TimerTask timerTaskFISCHER = new RESTCollectorMongo(Mh.FISCHER);
        TimerTask timerTaskTEST = new RESTCollectorMongo(Mh.TEST);
        TimerTask timerTaskWALKER = new RESTCollectorMongo(Mh.WALKER);
        TimerTask timerTaskZWISSIG = new RESTCollectorMongo(Mh.ZWISSIG);

        //running timer task as daemon thread
        Timer timerTEST = new Timer(true);
        //starttime = jetzt | delay 20min = 1200000ms
        timerTEST.schedule(timerTaskTEST, startTime, 1200000);
        LOG.info("Timertask MH:TEST gestartet");

        //300000ms = 5min
        Timer timerFISCHER = new Timer(true);
        timerFISCHER.schedule(timerTaskFISCHER,
            new Date(startTimeInMillis + (1 * 300000)), 1200000);
        LOG.info("Timertask MH:FISCHER gestartet");

        Timer timerWALKER ...

        //Main Thread laufen lassen
        while(true){
            try {
                Thread.sleep(10000);
            } catch (InterruptedException ex) {
                ...
            }
        }
    }
}
```

Auszug aus der RESTCollectorMongo Klasse, welche die Imports beinhaltet:

Listing 4: Ausschnitt RestCollectorMongo

```
...
public class RESTCollectorMongo extends TimerTask implements RESTCollector {
    ...
}
```

```

public void run() {
    //Einstiegspunkt fuer den TimerTask
    LOG.info("Timer task started at:" + new Date());
    boolean importsOK = completeImport(); //imports ausfuehren
    if (importsOK) {
        LOG.debug("Alle Imports abgeschlossen");
    } else {
        LOG.warn("Probleme waehrend Import aufgetreten");
    }
    LOG.info("Timer task finished at:" + new Date() + );
}

...
public boolean completeImport() {
    //4 imports machen und am schluss bool verwerten
    boolean i1 = importAllMoebelhauser();
    boolean i2 = importAllOrders();
    boolean i3 = importAllDeliveries();
    boolean i4 = importAllProducts();

    try {
        dateTimeFix();
    } catch (ParseException ex) {
        LOG.error("DateTimeFix fehlgeschlagen ,
            inkonsistente Datumstypen in der Datenbank!");
    }
    return (i1 && i2 && i3 && i4);
}

...
}

```

Der Collector holt eigenständig die Daten vom REST Server und aktualisiert die Datenbank falls notwendig. Sollten Daten aus der DB gelöscht werden, werden diese beim nächsten Import automatisch wieder eingespielt.

## 6.4 Client

Der Client ist als ausführbares Java JAR implementiert. Dieser liefert ein CLI (Command Line Interface) auf der Basis vom JCommander<sup>8</sup> Framework. Dieses übernimmt die Verarbeitung von Parametern. Man kann somit definieren, welcher Parameter welche Aktion (Methode) aufruft. Das schöne an dieser Lösung ist, dass man sich nicht im Detail um das parsen und verarbeiten der CLI Commands kümmern muss (z.B. mittels unschönen Switch-Case Blöcken!). Ausserdem ist JCommander ein sehr flexibles Framework, mit welchem neue Commands ohne grossen Aufwand hinzugefügt werden können.

Die Parameter werden Annotation Based gesetzt:

Listing 5: Ausschnitt JCommander Client Konfiguration

```

@Parameters(separators = "=") //leertaste funktioniert defaultmaessig
public class RMICLIParameters {

    ...
}

```

---

<sup>8</sup><http://jcommander.org/>

```

@Parameter(names = {"-h", "--help"},
            help = true, //isHelp —> ja
            description = "Zeigt alle Commands, Parameter des CLI Clients
an")
private boolean help;

@Parameter(names = {"-r", "-rmiserver"},
            validateWith = _ServerIP_Validator.class,
            description = "IP des RMIServers")
private String rmiServerIP = "10.177.1.181";

@Parameter(names = {"-d", "-databaseIP"},
            validateWith = _ServerIP_Validator.class,
            description = "IP des Datenbankservers (MongoDB)"
)
...

//Auszug aus der ServerIP-Validator Klasse:
...
public class _ServerIP_Validator implements IParameterValidator {

@Override
public void validate(String name, String value)
throws ParameterException {
    boolean isValid = InetAddressValidator.getInstance().isValid(value);
    if (!isValid) {
        String message = "Keine gueltige IP Adresse! (" + value + ")";
        throw new ParameterException(message);
        ...
    }
}
}
}

```

Interessant ist auch die Möglichkeit, eigene Validator oder Converter Klassen mitzugeben, welche die Eingaben gleich validieren

Listing 6: Ausschnitt JCommander Converter Konfiguration

```

@Parameter(names = {"-m", "-moebelhersteller"},
            required = true,
            converter = _moebelhersteller_Converter.class,
            description = "Angabe des zu bearbeitenden Moebelherstellers"
)
...

//Auszug aus der Moebelhersteller-Converter Klasse:
public class _moebelhersteller_Converter
implements IStringConverter<MoebelherstellerEnum> {

@Override
public MoebelherstellerEnum convert(String value) {
    switch (value) {
        case "FISCHER":
        case "Fischer":
        case "fischer":

```

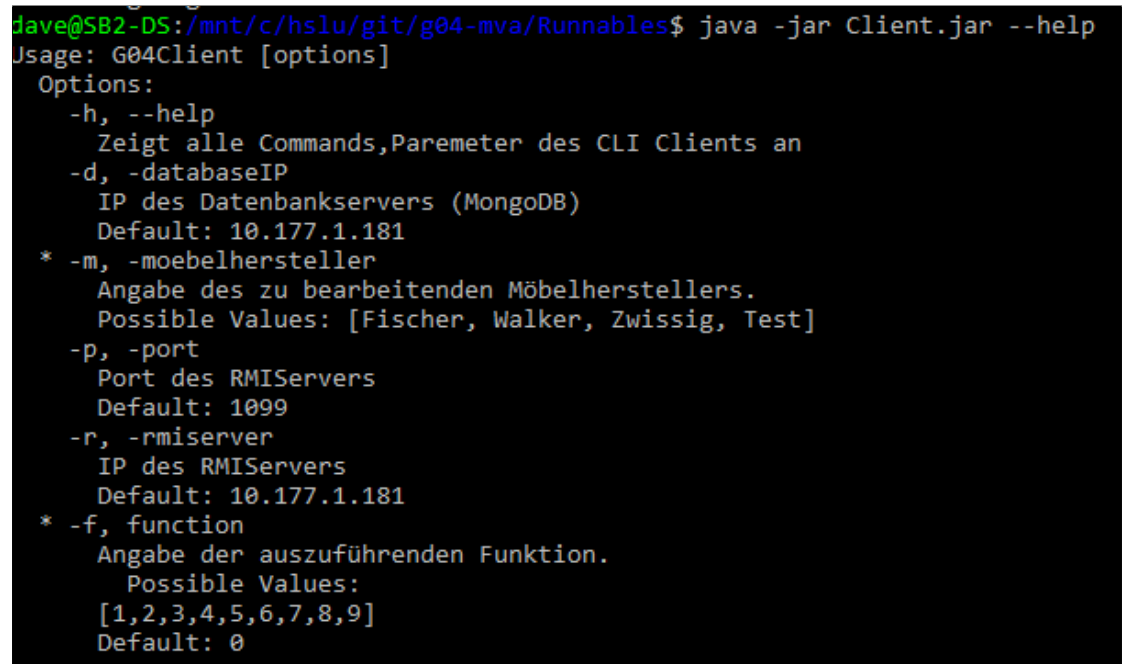
```
        return MoebelherstellerEnum.FISCHER;
    ...
```

Für die Kompilierung eines ausführbaren JARs mit allen Dependencies haben wir das "maven-assembly-plugin" verwendet (Source: <https://goo.gl/DvW5Qg>).

Nachdem das JAR mit den Abhängigkeiten kompiliert wurde, kann es wie folgt gestartet werden:

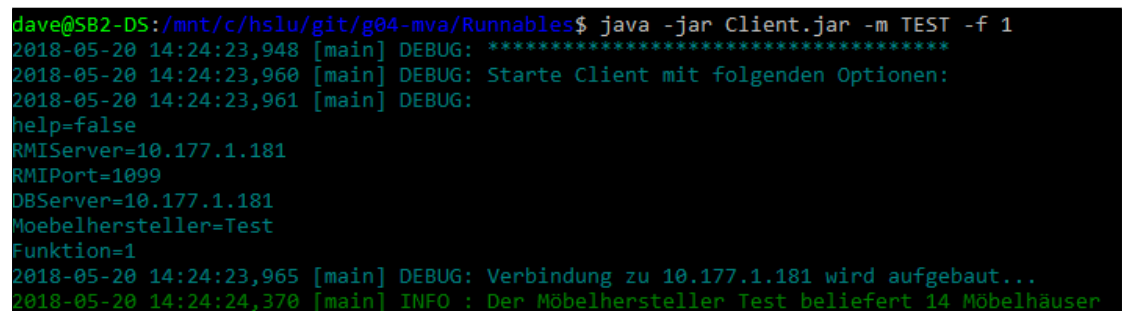
```
java -jar Client.jar --help
```

Hier einige Auszüge aus dem CLI:



```
dave@SB2-DS:/mnt/c/hslu/git/g04-mva/Runnables$ java -jar Client.jar --help
Usage: G04Client [options]
Options:
  -h, --help
      Zeigt alle Commands, Parameter des CLI Clients an
  -d, -databaseIP
      IP des Datenbankservers (MongoDB)
      Default: 10.177.1.181
  * -m, -moebelhersteller
      Angabe des zu bearbeitenden Möbelherstellers.
      Possible Values: [Fischer, Walker, Zwissig, Test]
  -p, -port
      Port des RMIServers
      Default: 1099
  -r, -rmiserver
      IP des RMIServers
      Default: 10.177.1.181
  * -f, function
      Angabe der auszuführenden Funktion.
      Possible Values:
      [1,2,3,4,5,6,7,8,9]
      Default: 0
```

Abbildung 14: CLI: Auszug HELP Command



```
dave@SB2-DS:/mnt/c/hslu/git/g04-mva/Runnables$ java -jar Client.jar -m TEST -f 1
2018-05-20 14:24:23,948 [main] DEBUG: *****
2018-05-20 14:24:23,960 [main] DEBUG: Starte Client mit folgenden Optionen:
2018-05-20 14:24:23,961 [main] DEBUG:
help=false
RMIServer=10.177.1.181
RMIPort=1099
DBServer=10.177.1.181
Moebelhersteller=Test
Funktion=1
2018-05-20 14:24:23,965 [main] DEBUG: Verbindung zu 10.177.1.181 wird aufgebaut...
2018-05-20 14:24:24,370 [main] INFO : Der Möbelhersteller Test beliefert 14 Möbelhäuser
```

Abbildung 15: CLI: Auszug Funktion 1 (Wieviele Möbelhäuser werden beliefert)

```
dave@SB2-DS:/mnt/c/hslu/git/g04-mva/Runnables$ java -jar Client.jar -m TEST -f 3
2018-05-20 14:24:45,761 [main] DEBUG: *****
2018-05-20 14:24:45,764 [main] DEBUG: Starte Client mit folgenden Optionen:
2018-05-20 14:24:45,764 [main] DEBUG:
help=false
RMIServer=10.177.1.181
RMIPort=1099
DBServer=10.177.1.181
Möbelhersteller=Test
Funktion=3
2018-05-20 14:24:45,769 [main] DEBUG: Verbindung zu 10.177.1.181 wird aufgebaut...
2018-05-20 14:24:46,025 [main] DEBUG: Möbelhaus: MH_BUCHELI_HORW | Durchschnitt-BW: 24431.5
2018-05-20 14:24:46,026 [main] DEBUG: Möbelhaus: MH_EGGER_EBLU | Durchschnitt-BW: 63441.0
2018-05-20 14:24:46,027 [main] DEBUG: Möbelhaus: MH_DIGA_EMME | Durchschnitt-BW: 17545.4
2018-05-20 14:24:46,028 [main] DEBUG: Möbelhaus: MH_MICA_IBAS | Durchschnitt-BW: 9890.0
2018-05-20 14:24:46,028 [main] DEBUG: Möbelhaus: MH_SCHU_ZHRI | Durchschnitt-BW: 11872.5
2018-05-20 14:24:46,029 [main] DEBUG: Möbelhaus: MH_TTIP_SPRE | Durchschnitt-BW: 33600.0
2018-05-20 14:24:46,029 [main] DEBUG: Möbelhaus: MH_PFIS_SUHR | Durchschnitt-BW: 12987.0
2018-05-20 14:24:46,030 [main] DEBUG: Möbelhaus: MH_CONF_SCHL | Durchschnitt-BW: 17910.666666666668
2018-05-20 14:24:46,031 [main] DEBUG: Möbelhaus: MH_INTE_PRAT | Durchschnitt-BW: 4866.5
2018-05-20 14:24:46,031 [main] DEBUG: Möbelhaus: MH_MAER_PFSZ | Durchschnitt-BW: 15035.0
2018-05-20 14:24:46,032 [main] DEBUG: Möbelhaus: MH_LIPO_EGER | Durchschnitt-BW: 12578.5
2018-05-20 14:24:46,032 [main] DEBUG: Möbelhaus: MH_HUBA_RTHI | Durchschnitt-BW: 5289.0
2018-05-20 14:24:46,033 [main] DEBUG: Möbelhaus: MH_FERR_HINW | Durchschnitt-BW: 21783.0
2018-05-20 14:24:46,034 [main] DEBUG: Möbelhaus: MH_MZMO_VOLK | Durchschnitt-BW: 0.0
```

Abbildung 16: CLI: Auszug Funktion 3 (Durchschnittlicher Bestellwert pro Möbelhaus)

```
dave@SB2-DS:/mnt/c/hslu/git/g04-mva/Runnables$ java -jar Client.jar -m TEST -f 8
2018-05-20 14:25:12,341 [main] DEBUG: *****
2018-05-20 14:25:12,343 [main] DEBUG: Starte Client mit folgenden Optionen:
2018-05-20 14:25:12,344 [main] DEBUG:
help=false
RMIServer=10.177.1.181
RMIPort=1099
DBServer=10.177.1.181
Möbelhersteller=Test
Funktion=8
2018-05-20 14:25:12,349 [main] DEBUG: Verbindung zu 10.177.1.181 wird aufgebaut...
2018-05-20 14:25:12,624 [main] INFO : ***Ausgabe Bestellungen über alle KWs des aktuellen Jahres***
2018-05-20 14:25:12,628 [main] INFO : Möbelhaus: 'MH_BUCHELI_HORW' | KW: '1' | AnzahlBestellungen: '2'
2018-05-20 14:25:12,630 [main] INFO : Möbelhaus: 'MH_BUCHELI_HORW' | KW: '6' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,632 [main] INFO : Möbelhaus: 'MH_DIGA_EMME' | KW: '4' | AnzahlBestellungen: '2'
2018-05-20 14:25:12,633 [main] INFO : Möbelhaus: 'MH_DIGA_EMME' | KW: '6' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,635 [main] INFO : Möbelhaus: 'MH_DIGA_EMME' | KW: '9' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,637 [main] INFO : Möbelhaus: 'MH_DIGA_EMME' | KW: '10' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,638 [main] INFO : Möbelhaus: 'MH_MICA_IBAS' | KW: '7' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,639 [main] INFO : Möbelhaus: 'MH_MICA_IBAS' | KW: '9' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,640 [main] INFO : Möbelhaus: 'MH_SCHU_ZHRI' | KW: '10' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,642 [main] INFO : Möbelhaus: 'MH_CONF_SCHL' | KW: '3' | AnzahlBestellungen: '2'
2018-05-20 14:25:12,643 [main] INFO : Möbelhaus: 'MH_INTE_PRAT' | KW: '4' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,644 [main] INFO : Möbelhaus: 'MH_MAER_PFSZ' | KW: '2' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,645 [main] INFO : Möbelhaus: 'MH_MAER_PFSZ' | KW: '6' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,647 [main] INFO : Möbelhaus: 'MH_LIPO_EGER' | KW: '2' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,647 [main] INFO : Möbelhaus: 'MH_HUBA_RTHI' | KW: '4' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,648 [main] INFO : Möbelhaus: 'MH_FERR_HINW' | KW: '3' | AnzahlBestellungen: '1'
2018-05-20 14:25:12,648 [main] INFO : Möbelhaus: 'MH_FERR_HINW' | KW: '9' | AnzahlBestellungen: '1'
```

Abbildung 17: CLI: Auszug Funktion 8 (Ausgabe Bestellungen über alle KWs des aktuellen Jahres)

## 7 Lessons Learned

### 7.1 Übersicht schaffen

Die Autoren haben sich zur Aufgabenstellung und den dazugehörigen Herausforderungen einige Gedanken gemacht und diese auf Basis eines Brainstormings geordnet. Anschließend haben sich die Autoren über den Systemaufbau unterhalten und diesen auf der nachfolgenden Abbildung festgehalten.

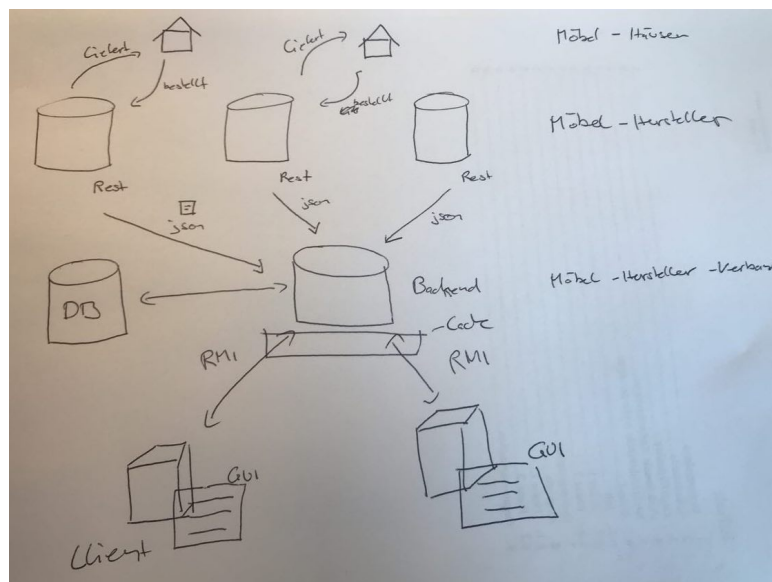


Abbildung 18: Systemskizze

Das festhalten und nachschlagen der so geschaffenen Systemübersicht hat die Zusammenarbeit wesentlich erleichtert, da so ein einheitliches Verständnis über die Aufgaben und dessen Teilaufgaben geschaffen werden konnte. Zudem konnte man die Übersicht beiziehen, sobald etwas unklar wurde.

### 7.2 Herausforderungen

#### 7.2.1 JSON, GSON, Jackson, was ist das eigentlich?

Zu Beginn des Projekts wurden hauptsächlich die REST Schnittstellen und die Daten, welche von diesen geliefert werden, analysiert.

Zu Beginn musste also zuerst verstanden werden, wie genau das JSON Format funktioniert, was JSON-Objekte und was JSON-Arrays sind.

Nachdem ein fundiertes Verständnis für das JSON Format geschaffen wurde, entschied man sich für die Verwendung des von Google entwickelten GSON<sup>9</sup> Frameworks zur Verarbeitung der Daten. Dieses wird auf diversen Plattformen im Internet gelobt für seine schnelle und intuitive Verarbeitung von grossen Datenmengen im JSON Format.

<sup>9</sup><https://github.com/google/gson>

Nachdem die Gruppe sich in die GSON Dokumentation eingelesen und sich an praktischen Übungen versucht hatte, wurde nach einiger Zeit die Verwendung des GSON Frameworks wieder abgebrochen. Das Framework ist schlicht zu komplex und 'überladen' für die Anforderungen des Projekts.

Die Gruppe Entschied sich für die Verwendung des in der Java Welt ebenfalls bekannten und beliebten Frameworks Jackson<sup>10</sup> für die Verarbeitung der JSON Daten.

Das Jackson Framework stellt alle benötigten Funktionen zur Verfügung. Außerdem ist die Verwendung intuitiv und das Framework ist nicht so überladen wie jenes von Google.

### 7.2.2 Architektur Entscheid - Datenbank

Auf der Suche nach einer komfortableren Lösung für die Verarbeitung von grossen Mengen an JSON Daten, entschieden sich die Autoren für den Einsatz einer NoSQL-Datenbank.

Sie entschieden sich relativ spontan für das Produkt "MongoDB", ohne sich detaillierter über die Tragweite dieses Entscheids Gedanken zu machen. MongoDB erlangte in den letzten Jahren stetig mehr Beliebtheit in der Community und eignet sich optimal für den Einsatz um grosse, unstrukturierte Datenmengen, wie wir sie von den REST Schnittstellen erhalten, zu verarbeiten.

Diese Entscheidung führte dazu, dass die Autoren sich nebst der Aufgabenstellung detailliert mit den technischen Möglichkeiten der MongoDB sowie des Mongo-Java-Drivers bekanntmachen mussten, was **nicht unwesentlich mehr Zeit** in Anspruch nahm.

In der Retroperspektive kann gesagt werden, dass der Einsatz von MongoDB das Projekt wohl noch etwas komplexer gemacht hat und somit nicht die einfachste Lösung darstellt. Nichtsdestotrotz, die Entwicklung mit MongoDB hat unser praktisches Wissen in Bezug auf NoSQL und dessen Anwendung enorm gesteigert.

Schlussendlich haben wir nun eine Lösung erarbeitet welche einen pragmatischen SOC (Separation of Concerns) Ansatz verfolgt. Die Verwendung einer zentralen Datenbank als Bindeglied zwischen den REST-Schnittstellen und dem CORE-System sorgt außerdem für eine professionelle Lösung, welche das Risiko einer Überlastung der Systeme oder ungewollte Manipulation der Datensätze auf ein Minimum reduziert.

### 7.2.3 Erarbeitung der funktionalen Anforderungen

Die Erarbeitung der funktionalen Anforderungen schien zu Beginn relativ einfach zu sein. In Kombination mit MongoDB dürfte das ganze ja kein Problem darstellen. Was zu Beginn so einfach schien, stellte sich aber zunehmend schwieriger dar.

**A01 - A02** Die ersten beiden Anforderungen konnten relativ schnell mit wenigen Zeilen Code gelöst werden. Schliesslich musste von der Datenbank nur die Anzahl von zwei verschiedenen Collection-Items zurückgegeben werden.

---

<sup>10</sup><https://github.com/FasterXML/jackson>

### Listing 7: A01 und A02

```
public class StartCollector
@Override
public int getMoebelhauser01() {
    return (int) this.mongoreader.ReadALL(this.moebelhauserCollection)
        .count();
}

@Override
public int getProductTypes02() {
    return (int) this.mongoreader.ReadALL(this.productsCollection).count();
}
```

**A03 - A07** Die Anforderungen drei bis sieben stellten dann doch etwas mehr Schwierigkeiten dar. MongoDB vereinfachte unsere Arbeiten nicht unbedingt, sondern veränderte die Art wie wir mit den Daten umgehen. Da wir mit 'Documents' arbeiten, was wiederum nur Container für JSON Objekte und Arrays darstellt, haben wir uns dafür entschieden direkt mit diesen zu arbeiten und nicht noch zuerst Objekte daraus zu erstellen. Das bringt unter anderem den Vorteil, das wir Dokumente einfach wieder in die DB zurückspeichern können (z.B. für Caching) und nicht gross mit Castings und Typumwandlungen arbeiten müssen.

Ein Teil der Funktionen ist daher immer ähnlich. Sobald mehrere Möbelhäuser angeschaut werden müssen kommt folgende Iteration zum Zuge:

### Listing 8: Auszug A07

```
///#1 Iteration ueber alle Moebelhaeuser
for (Document mh_doc : alleMoebelHauser) {
    String moebelhausCode = mh_doc.get("moebelhausCode").toString();

    BasicDBObject searchQuery = new BasicDBObject();
    searchQuery.put("datum", new BasicDBObject("$gte", Von)
        .append("$lt", Bis));
    FindIterable<Document> matchings = findIterableBuilder
        (("Bestellungen_" + moebelhausCode + "_" + this.moebelherstellerEnum
            .toString()), new Document("bestellungPositionListe.anzahl", 1)
            .append("bestellungPositionListe.produktTyp.id", 1), searchQuery);
    ///#2 Iteration des Unterarrays Bestellungen
    for (Iterator<Document> bestellung_it = matchings.iterator();
        bestellung_it.hasNext();) {
        Document maindoc = bestellung_it.next();
        List<Document> bestellPosition =
            (List<Document>) maindoc.get("bestellungPositionListe");
        ///#3 Iteration des Unterarrays Bestellposition
        for (Document position_doc : bestellPosition) {
            Integer anzahl = position_doc.getInteger("anzahl");
            ///#4 Objekt Produkttyp auslesen
            Document produktTyp = (Document) bestellPosition.get(0)
                .get("produktTyp");
            Integer produktTyp_id = produktTyp.get("id", Integer.class);
            if (!produktID_Anzahl_Map.containsKey(produktTyp_id)) {
                produktID_Anzahl_Map.put(produktTyp_id, anzahl);
            } else {
                produktID_Anzahl_Map.put(produktTyp_id,
                    produktID_Anzahl_Map.get(produktTyp_id) + anzahl);
            }
        }
    }
}
```



```

    + anzahl);
    }
  }
}

```

**A08 - A09** Die letzten beiden Anforderungen hatten noch die zusätzliche Bedingung, das Caching Mechanismen verwendet werden, weil diese Abfragen anscheinend viel Zeit benötigen. Dank der Verwendung von MongoDB hatten die Autoren zwar keine Schwierigkeiten mit der Performance, setzten jedoch trotzdem das Caching um.

Um die Anforderung umzusetzen werden zwei weitere 'Cache' Collections für A08 und A09 erzeugt. Bei der ersten Verwendung wird die Abfrage normal durchgeführt und am Ende eine separate 'Cache'-Collection mit den effektiven Resultaten (ohne Berechnung) erzeugt.

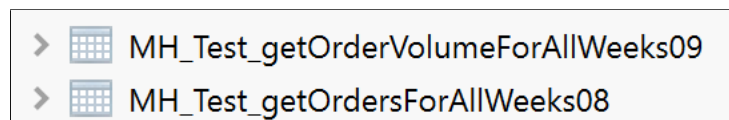


Abbildung 19: Caching Collections

Beim zweiten Abfragen wird jeweils direkt die Cache-Collection ausgelesen und zurückgegeben.

Nun stellte sich aber die Frage 'Wann ist die Cache-Collection veraltet'? In der Praxis würde man hier Prozesse mit den Lieferanten vereinbaren. Beispielsweise müssten die Bestellungen einer Kalenderwoche (KW) immer spätestens in der nächsten KW erfasst sein.

Dann wäre der Cache Algorithmus relativ einfach, alle Dhttp://80.67.147.28:3030/project/5afb060a261adb00 die älter als zwei Wochen sind, wären 'korrekt'. Somit müsste nur noch geprüft werden, ob die aktuelle KW und die in der Cache-Collection nicht übereinstimmen.

Beim Tomcat-JSON Server schienen Daten der Vergangenheit ab und zu zu ändern (oder wir haben uns das eingebildet), jedenfalls würde dieser Umstand dafür sorgen, dass viel ausgeklügeltere Caching-System verwendet werden müssten, oder das Thema Caching ganz gestrichen wird, da dies mit MongoDB keine grossen Probleme darstellt.

Wir haben uns für die pragmatische Variante entschieden, unter der Annahme dass die vergangenen KWs immer korrekt sind und nicht mehr verändert werden.

Listing 9: Hilfsklasse 'checkIfCollectionAgeUp2date'

```

public boolean checkIfCollectionAgeUp2date(JSONArray persistedArray) {
    // Aktuelle KW berechnen
    Calendar now = Calendar.getInstance();
    int aktuelleKW = now.get(Calendar.WEEK_OF_YEAR);
    int dbKW = 0;

    for (int i = 0; i < persistedArray.length(); i++) {
        JSONObject job_ArrayItem = persistedArray.getJSONObject(i);
        JSONArray jar_Bestellungen = job_ArrayItem
            .getJSONArray("Bestellungen");
        for (int j = 0; j < jar_Bestellungen.length(); j++) {

```

```

        JSONObject job_b = jar_Bestellungen.getJSONObject(j);
        dbKW = job_b.getInt("KW");
        LOG.debug("KW: " + dbKW);
        if (dbKW >= aktuelleKW) {
            LOG.debug("KW ist up2date, Collection ist aktuell");
            LOG.debug("AktuelleKW: " + aktuelleKW +
                " | dbKW: " + dbKW);
            return true;
        }
    }
    return false;
}

```

## Typsichere Preise

Bei der Erarbeitung der funktionalen Anforderungen sind wir noch auf einen speziellen Umstand gestossen.

Alle Preise von Produkten sind vom Typ Integer, ausser der Preis von Einem Produkt. Dieses hat den Typ Float (bzw. Double).

Dieser Umstand hat beim Testing zu merkwürdigen Fehlern geführt. Weil das Problem verschachtelt war (der Fehler trat nur beim umwandeln von Werten aus der Datenbank auf) musste die DB-Collection genau angeschaut werden:

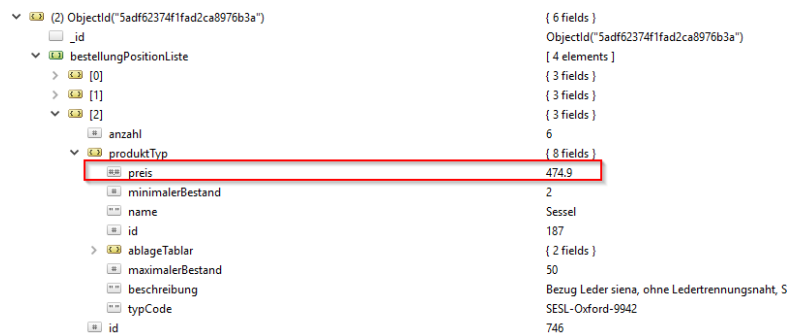


Abbildung 20: Der Übeltäter

Dieser Umstand, unter der Berücksichtigung, dass es durchaus Sinn macht Preise im Typ Float zu speichern, führte dazu, dass wir folgende Hilfsklasse einführten. Diese prüft beim Einlesen der Daten aus der DB den Typ:

Listing 10: Hilfsklasse 'getPreistypeSafe'

```

private Double getPreistypeSafe(Document sourceDoc) {
    Double preis;
    //Pruefen ob der Typ nicht double ist und dann konvertieren!
    if ((sourceDoc.get("preis")) instanceof Integer) {
        //Int gefunden
        preis = sourceDoc.get("preis", Integer.class).doubleValue();
    } else {
        //Double gefunden
        preis = sourceDoc.get("preis", Double.class);
    }
    return preis;
}

```

}

#### 7.2.4 Daily struggle with Jenkins

Jenkins ist ein Werkzeug, welches die Autoren zu schätzen lernten. Es hat auf viele Fehler oder Unschönheiten hingewiesen. Leider gab es auch Momente, in denen Jenkins sich einfach Grundlos beklagte.

Vorallem mit Kommentaren führten die Autoren mit Jenkins einen erbitterten Kampf. Denn Jenkins interpretiert sämtliche Zeichen aus Kommentaren, Symbole wie '<,>,-' teilweise als Code. Dieser kann nicht ausgeführt werden (vor allem Kommentare wie 'Xy führt zu -> Resultat abc'), hier wurde von Jenkins wohl probiert Lambdas auszuführen.

### 7.3 RMI Testing

Die Autoren stellten auch fest, dass Testing von Software, welche entferne Ressourcen benötigt, nicht trivial ist. Es wurden etliche Beispiele von Mocking-Technologien konsultiert und Blogposts gelesen. Schlussendlich konnten aber kaum sinnvolle und zweckbringende Tests entwickelt werden.

Schlussendlich wurde auf das automatisierte Testing von Code, welcher Netzwerkschnittstellen verwendet, aufgrund des Zeitdrucks mehrheitlich verzichtet und 'von Hand' getestet.

### 7.4 Kohäsion und Kopplung

In der Retroperspektive zum Projekt fragen sich die Autoren, gegen wieviele Design Prinzipien sie wohl verstossen haben.

Der Ansatz Test-Frist beispielsweise wurde zu Beginn versucht, konnte jedoch nicht vollkommen durchgezogen werden. Die Autoren stellten fest, dass sie vor dem Design der Funktionen schon darauf achteten, dass diese einfach zu testen sind. Dies hat beim anschließenden implementieren der Tests sehr geholfen.

Das Hauptziel war es, die beiden Prinzipien hohe Kohäsion und lose Kopplung so gut wie möglich umzusetzen.

Mit insgesamt 33 Klassen, Interfaces und Enumeratoren wurde das Ziel der hohen Kohäsion sicherlich erreicht. Es wurde darauf geachtet, kleine, wohldefinierte Klassen mit überschaubarem Funktionsumfang zu entwickeln, welche wiederverwendet werden könnten.

Ausser der Klasse 'FurnitureManufacturer' gibt es kaum Klassen, welche als Überladen zu bezeichnen sind. Da dort jedoch die gesamte 'Logik' der funktionalen Anforderungen implementiert wurde, ist diese entsprechend gross geworden. Eine mögliche Option der Verkleinerung wäre es gewesen, die einzelnen Anforderungen in separaten Klassen umzusetzen. Dafür hätte jedoch auch das Interface angepasst werden müssen. Die Autoren haben sich aber bewusst dagegen entschieden, da Sie grundsätzlich mit dem Design sehr zufrieden waren.

Durch die Aufteilung in die vier Module 'Server,Collector,Datenbank und Client', wurde ebenfalls eine lose Kopplung erreicht. Lediglich innerhalb der einzelnen Module gibt es eine etwas stärkere Kopplung. Die Module Collector und Server sind aber beispielsweise komplett unabhängig voneinander.

## 7.5 Fazit

Die Autoren sind mit ihrer Lösung sehr zufrieden. Alle Anforderungen konnten umgesetzt werden. Zudem konnte viel über die Modularisierung gelernt und Erfahrungen im Projektmanagement gesammelt werden. Auch die gelernten Technologien wie 'MongoDB' und 'Docker' sind Kompetenzen, welche sicherlich noch oft eingesetzt werden können.

Die Motivation wurde jedoch durch die Tatsache gedämpft, dass diese Arbeit nicht in die Bewertung der Modulabschlussprüfung einfließt. Dies ist nicht nachvollziehbar, denn in anderen Modulen (bspw. DBS) ist dies möglich.

Nach der Meinung der Autoren sollte sich der Aufwand auszahlen. Vorallem, wenn weit mehr Stunden investiert werden, als von den für das Modul erhaltenen sechs Credits gefordert werden. Unsere Lösung ist definitiv eine Lösung, in welche viel mehr Zeit als nötig investiert wurde.

Zum Schluss muss jedoch auch erwähnt werden, dass die hohe Freiheit und Flexibilität der Aufgabenstellung Spass gemacht hat. Zudem war das aufgebaute Szenario an mögliche reale Anwendungen angelehnt. Die Autoren können sicherlich einiges in diesem Projekt gelerntes direkt bei ihren Arbeitgebern implementieren und umsetzen.