RHEINISCHE FACHHOCHSCHULE KÖLN

University of Applied Sciences

Fachbereich: Wirtschaft & Recht

Studiengang: Business Information Management (B.Sc.)



**Dokumentation zum Laborprojekt**

Application Development 2  
Mini-CRM – verteilte Java-Anwendung mit JDBC und RMI

vorgelegt von: Maik Godinho (Mat.-Nr.: BWI2131010)

Denis Kündgen (Mat.-Nr.: BWI2131014)

Nina Ziegler (Mat.-Nr.: BWI2131008)

Dozent: Herr Dipl.-Kfm. Matthias Bohnen

Wintersemester 2014

Inhaltsverzeichnis

[1 Einleitung 1](#_Toc410322460)

[2 Vorgehensweise 1](#_Toc410322461)

[3 Anforderungsanalyse 2](#_Toc410322462)

[4 Konzeptionierung 4](#_Toc410322463)

[4.1 Entitäten 4](#_Toc410322464)

[4.2 Aufbau der Anwendung 5](#_Toc410322465)

[4.3 Speichern von Informationen 6](#_Toc410322466)

[5 Technische Umsetzung 7](#_Toc410322467)

[5.1 „mock“-Klassen 7](#_Toc410322468)

[5.2 Observer Pattern 7](#_Toc410322469)

[5.3 Factory Pattern 9](#_Toc410322470)

[6 UML-Klassendiagramme der Anwendung 10](#_Toc410322471)

[6.1 Client 10](#_Toc410322472)

[6.2 ContactService 11](#_Toc410322473)

[6.3 AppointmentService 12](#_Toc410322474)

[6.4 ConnectionManager 13](#_Toc410322475)

[6.5 Gesamt 13](#_Toc410322476)

[7 Eingereichte Ergebnisse 14](#_Toc410322477)

[8 Installationsanleitung 14](#_Toc410322478)

[9 Fazit 15](#_Toc410322479)

[Darstellungsverzeichnis 16](#_Toc410322480)

# Einleitung

Ziel dieser Projektarbeit ist die Entwicklung eines lauffähigen Moduls, welches einen Use Case einer komplexen verteilten Java-Anwendung abbildet. Dabei soll vor allem auf Architektur und Klassenverteilung geachtet werden.

Voraussetzung ist das Trennen der Applikation in ein Client-Server-Modell. Mittels Remote Method Invocation (RMI) sollen serverseitige Objekte und Methoden auch auf dem Client zur Verfügung stehen. Das dauerhafte Speichern von Objekten soll mit Hilfe einer Persistenzschicht geschehen. Diese bietet mit Hilfe von JDBC die Möglichkeit, Daten dauerhaft in einer Datenbank abzulegen. Zudem werden die Daten über XML auch weiteren Applikationen zur Verfügung gestellt.

# Vorgehensweise

Zur Entwicklung der Anwendung waren unterschiedliche Aufgaben erforderlich. In gemeinsamen Workshops bzw. Skype-Konferenzen erfolgte die Verteilung und Abstimmung der einzelnen Aufgabenpakete.

* **Entwicklung eines Grobkonzepts**(gemeinsam)
* **Erstellung eines Datenbankmodells**   
  (Maik Godinho)
* **Erstellung des Clients**  
  (Nina Ziegler, Denis Kündgen)
* **Aufbau der Server-Architektur**  
  (Maik Godino)
* **XML-Ausgabe**   
  (Nina Ziegler)
* **Anwendungslogik und Datenbankanbindung**  
  (Maik Godinho)
* **Erstellung der Präsentation**   
  (Denis Kündgen)
* **Erstellung der vorliegenden Dokumentation**  
  (Denis Kündgen)

# Anforderungsanalyse

Anhand der beschriebenen Aufgabenstellung konnte bereits eine Vielzahl der Anforderung bestimmt werden.

Als **nichtfunktional** sind hier folgende Anforderungen aufzuführen:

* Client-Server-Architektur
* Nutzung von Server-Objekten im Client (RMI)
* Nachvollziehbare Verwendung von Konstrukten und verständliche Auscodierung
* Bereitstellen von Testfällen oder einer Testumgebung

Als **funktional** sind hier folgende Anforderungen aufzuführen:

* Umsetzung eines Use Cases einer komplexen Java-Anwendung
* XML-Ausgabe von Objekten
* Anbinden einer Datenbank via Java Database Connectivity

Als komplexe Java-Anwendung wurde ein Customer Relationship Management-System gewählt. Customer Relationship Management-Systeme (im folgenden CRM-Systeme) verfolgen als primäres Ziel die Dokumentation und Verwaltung von Kundenbeziehungen. Dies wird häufig durch Relationen von Entitäten abgebildet. Mehrere „Vorgänge“, welche bspw. der Entität „Kontakt“ zugeordnet werden, bilden in chronologischer Sortierreihenfolge eine Historie zum jeweiligen Kontakt.

Aus der komplexen Anwendung eines CRM-Systems, wurden zunächst diverse Use Cases identifiziert (siehe Abbildung 1 – Use Case Diagramm). Da die Anforderung sich lediglich auf die Realisierung eines Use Cases beschränkt, wurden weitere Use Cases eines CRM-Systems vernachlässigt. Die Abbildung 1 zeigt, in blauer Markierung, die Use Cases, welche architektonisch in der hier beschriebenen Anwendung berücksichtigt wurden. Rot markierte Use Cases dienen als ergänzende Beispiele für ein CRM-System, finden im weiteren Verlauf jedoch keine Berücksichtigung im Projekt.

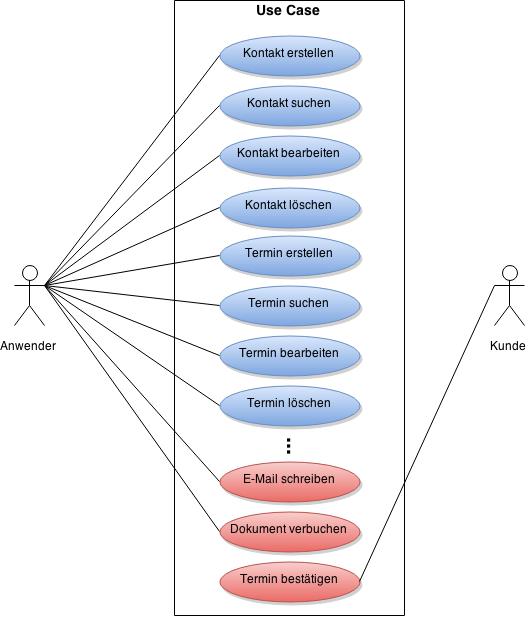


Abbildung 1 - Use Case Diagramm

In der hier beschriebenen Anwendung soll es, zunächst konzeptionell, möglich sein, Kontakte und Termine zu erfassen, zu suchen, zu bearbeiten und zu löschen. Termine werden zudem einzelnen Kontakten zugeordnet, sodass im späteren Verlauf alle Termine zu einem Kontakt ausgegeben werden können.

Als konkreten Anwendungsfall, zur Erfüllung der Anforderung „Programmierung eines Use Cases“, wurde der Use Case „Kontakt anlegen“ gewählt. Alle Weiteren werden jedoch zumindest architektonisch berücksichtigt.

# Konzeptionierung

## Entitäten

Zur Konzeptionierung der Anwendung wurden zunächst die beiden Entitäten „Contact“ (zur Speicherung von Kontaktinformationen) und „Appointment“ (zur Speicherung von Termininformationen) identifiziert. Da Informationen zu einer Person in der Realität unabhängig von seinem Wohnort existieren, wurden Informationen über den Wohnort des Kontakts aus der Entität „Contact“ extrahiert und in die Entität „Adress“ überführt.

Die Drei Entitäten, sowie Ihre Zusammenhänge werden im folgenden UML-Diagramm dargestellt.

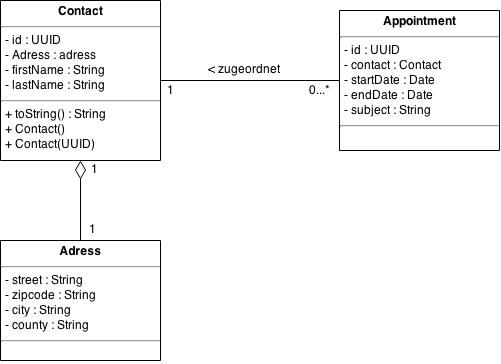


Abbildung 2 - UML-Klassendiagramm der Entitäten

Die Darstellung zeigt, dass eine Adresse jeweils einem Kontakt zugeordnet ist und ein Kontakt genau eine Adresse besitzt. Die Aufteilung in verschiedene Klassen wirkt dadurch vorerst überflüssig. Hier soll jedoch erwähnt werden, dass konzeptionell daran gedacht wurde, in späteren Versionen das Modul so auszubauen, so dass verschiedene Adressen einem Kontakt zugeordnet werden können (bspw. Privatadresse, Firmenadresse).

Die Methode toString() in der Klasse Contact dient zur gemeinsamen Ausgabe von Vor- und Zunamen eines Kontaktes. Bei der Methode contact() handelt es sich um eine sogenannte überladene Methode. Wird eine UUID beim Aufruf dieser Methode übergeben, so wird diese gesetzt. Andernfalls wird eine neue UUID generiert und gesetzt. So kann die gleiche Methode, sowohl für das generieren eines neuen Kontakt-Objekts, als auch beim Einlesen eines bestehenden Kontaktes in ein Objekt, verwendet werden.

Der oberen Darstellung sind zudem folgende Punkte hinzuzufügen:

* Zur späteren Verifizierung beim Deserialisieren von gespeicherten Daten wurden in allen Klassen das Attribut „serialVersionUID“ mit einem Wert versehen.
* Alle Klassen bieten, neben den angegebenen Methoden, die Standardmethoden zum Schreiben und Lesen von Attributswerten als öffentliche Methoden.

## Aufbau der Anwendung

Als nichtfunktionale Anforderung wurde die Verwendung einer Client-Server-Architektur vorgegeben. Mittels RMI sollen Serverobjekte auch im Client verwendbar sein. Aus diesem Grund wurde die Anwendung zunächst in zwei Module aufgeteilt.

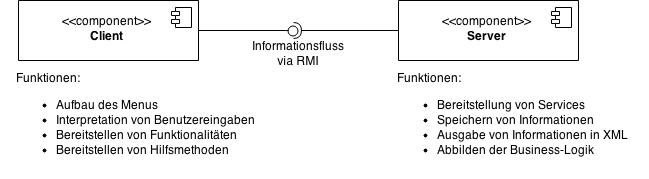


Abbildung 3 – Server-Client-Architektur

Der Client dient primär als Schnittstelle zum Benutzer. Hier werden Menus generiert, Eingaben interpretiert und die Funktionen zur Verfügung gestellt. Die jeweiligen Funktionen nutzen dann wiederum Objekte, welche vom Server zur Verfügung gestellt werden und können dort Methoden, bspw. für das Abrufen oder Speichern von Informationen, ausführen.

Der Server stellt via Interfaces diverse Services zur Verfügung. Diese ermöglichen die Kommunikation des Clients mit dem Server. Die Services verwalten intern die Business-Logik, in der beispielsweise einem Termin ein Kontakt-Objekt zugeordnet wird. Zudem obliegt dem Server die dauerhafte Speicherung von Informationen in einer Datenbank oder XML-Dateien.

Als weitere Anforderung wurden das Halten von Informationen in einer Datenbank mit Hilfe der Datenbankschnittstelle JDBC, sowie die Ausgabe per XML genannt. Für diese Anforderung bietet es sich an, den Aufbau des Servers in zwei Module zu teilen und eine dreischichtige Anwendung zu generieren.

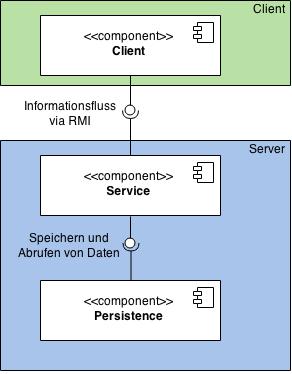


Abbildung 4 – konzeptioneller Modulaufbau Server-Client

## Speichern von Informationen

Das Speichern und Holen von Informationen wird durch die Persistenzsschicht übernommen. Diese regelt den Datenfluss über die Datenbankschnittstelle JDBC in eine Datenbank oder über eine XML-Datei. Folgende Skizze beschreibt den konzeptionellen Aufbau der Persistenzschicht.

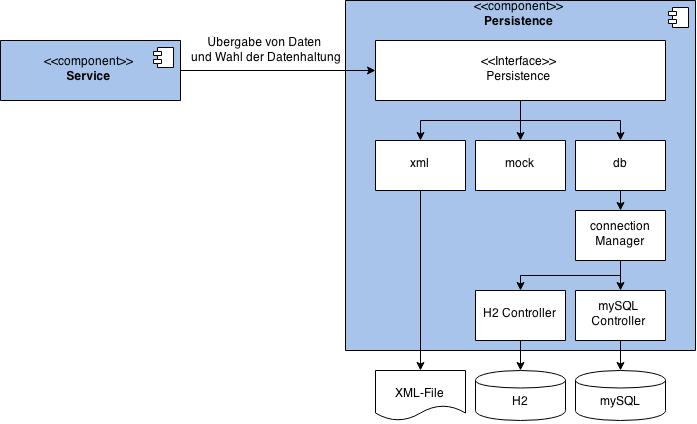


Abbildung 5 – Aufbau der Persistenzschicht

Wie in Abbildung 5 dargestellt, werden die Daten über den jeweiligen Service an die Schnittstelle der jeweiligen Persistenzschicht weitergereicht. Neben den Objektinformationen wird zudem im Service deklariert, welche Speichermethode angesprochen werden soll. Dabei werden zunächst drei Klassen zur Verfügung gestellt, welche das Persistence-Interface nutzen können.

Die XML-Klasse beinhaltet alle Methoden zur Serialisierung oder Deserialisieren von Daten eines XML-Files. Diese Klasse nutzt die JAVA-Klassen aus java.io für den Umgang des XML-Files in einem Document-Objekt.

Die mock-Klasse dient als virtuelles Hilfsobjekt. In diesem Objekt sollen Beispieldaten bereitgehalten werden, um bereits vor Realisierung der beiden Speichermethoden Clientseitig Anfragen gegen den Server zu testen.

Die Klasse „db“ soll den Umgang mit Datenbanken realisieren. Hier sollen mit Hilfe eines connectionManagers via JDBC die Datenbanken H2 und mySQL ansteuerbar sein. Die folgende Grafik zeigt den Aufbau der Datenbank.

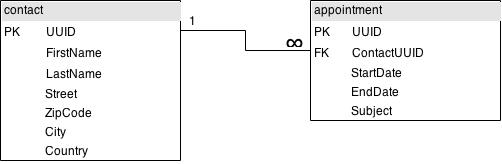


Abbildung 6 - Datenbankmodell

# Technische Umsetzung

Die technische Umsetzung erfolgte in mehreren Etappen. Grundlegend für den Start der Programmierung war der gemeinsame Workshop am 20.12.2014. Hier wurde, neben der Konzeptionierung des Projekts, sowohl die mySQL-Datenbank angelegt, als auch die Programmierung gestartet. Zur Vereinheitlichung von Dokumentation und Struktur wurde sich, zur Programmierung der Anwendung, auf die Software eclipse geeinigt.

Im weiteren Projektverlauf konnten die Teilnehmer eigenständig das jeweils zugeteilte Modul weiterentwickeln. Über die kostenlose Software GitHub, wurde ein gemeinsames Projekt erstellt. Die Software ermöglichte den ständigen Austausch und das Aktualisieren von Sourcecodes der jeweils anderen Gruppenteilnehmer.

Besonders zu erwähnen sind, neben der Nutzung einer Client-Server-Architektur, drei weitere Komponenten, die innerhalb Projekts umgesetzt wurden. Im Folgenden werden diese Komponenten einzeln erläutert.

## „mock“-Klassen

Während der Erstellung der Server-Komponente wurden gleichzeitig die Arbeiten am Client begonnen. Um die Programmierarbeiten des Clients immer wieder zu testen, wurden Beispielobjekte vom Server benötigt. Da das Servermodul jedoch noch nicht vollständig umgesetzt war, wurden sogenannte „mock“-Klassen implementiert. Mock-Klassen bilden während der Laufzeit ein mock-Objekt. Dieses beinhaltet wesentliche Daten und Methoden einer später auszuprogrammierenden Klasse und bietet die Möglichkeit, komplexe Module vorerst vereinfacht darzustellen.

Mittels den mock-Objekten, wurden dem Client somit Beispieldaten vorgegaukelt. Dies machte es möglich, schrittweise die Logik zu implementieren und trotzdem Daten anderen Modulen für Tests zur Verfügung zu stellen.

## Observer Pattern

Innerhalb des Clients wurde in der Architektur ein sogenanntes Observer Pattern implementiert. Observer Pattern stellen ein Entwurfsmuster in der Softwareentwicklung dar. Durch Ihren Einsatz ist es möglich bei einer Zustandsänderung eines Objekts andere Objekte zu benachrichtigen. Dies geschieht mit Hilfe eines sogenannten Publishers, der eine Schnittstelle anbietet, an der sich sogenannte Subscriber an- und abmelden können. Ändert sich der Zustand eines Objektes, so informiert dieses alle Subscriber über deren Aktualisierungsschnittstelle.

Die Ein- und Ausgabe des Clients wurde innerhalb eines Konsolenmenus realisiert. Dieses gibt dem Benutzer eine Übersicht über anwendbare Funktionen und leitet Ihn durch die Eingabe von Daten. Vorerst gelangt der Benutzer in das Menu1, in der die jeweils zur Verfügung stehenden Kategorien gelistet werden.

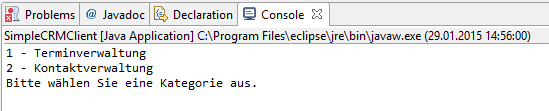


Abbildung 7 – Clientmenu 1

Wählt der Benutzer eine Kategorie durch Eingabe der vorangestellten Nummer, so wird gelangt er in das Menu der jeweils gewählten Kategorie.

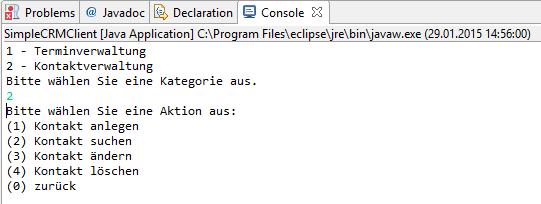


Abbildung 8 – Clientmenu 2

Bei der Auflistung der Module und dem Laden des zweiten Menus kommt das oben beschriebene Observer Pattern zum Einsatz. Anhand folgender Grafik soll der Ablauf beschrieben werden.

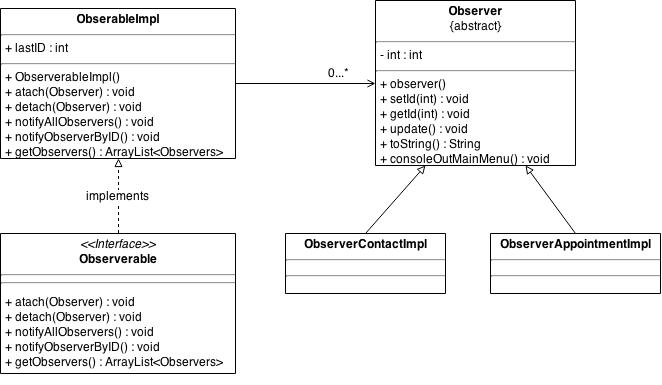


Abbildung 9 – UML Observer Pattern

Zunächst wird ein Interface „Observerable“ zur Verfügung gestellt. Dieses wird von der Klasse „ObserverableImpl“ implementiert. Diese gilt fortwährend als Publisher.

Gleichzeitig erben die Klassen „ObserverContactImpl“ und „ObserverAppointmentImpl“ von der abstrakten Klasse „Observer“. Dieser bietet später die Aktualisierungsschnittstelle zu den beiden genannten Subscribern.

Über die Main-Klasse „SimpleCRMClient“ werden die Subscriber am Publisher einmalig angemeldet. Über den Publisher kann zunächst abgefragt werden, welche Subscriber sich bei Ihm angemeldet haben. Die Ausgabe der angemeldeten Subscriber erzeugt das erste Clientmenu (siehe Abbildung 7) mit einer danach folgenden Aufforderung an den Benutzer. Wählt der Benutzer nun eine Kategorie, in dem er eine Zahl (Subscriber-ID) eingibt, so wird der Subscriber mit dieser ID benachrichtigt und erzeugt somit das zweite Clientmenu (siehe Abbildung 8).

Dies bildet zwar nicht den klassischen Einsatz eines Observer-Patterns ab, in dem alle Subscriber über die Änderung im Publisher benachrichtigt werden. Es bietet jedoch die Möglichkeit, zukünftig weitere Module im Client einfach hinzuzufügen. Module können so einfach am Publisher registriert werden ohne die Logik oder die Menustruktur verändern zu müssen.

## Factory Pattern

Zur Bereitstellung der einzelnen Services wurden sogenannte Factory Pattern eingesetzt. Diese „virtuellen Konstruktoren“ werden verwendet um über eine Schnittstelle einer abstrakten Klasse oder eines Interfaces neue Objekte zu erzeugen, die sich jedoch nicht selbst, sondern Unterklassen instanziieren, welche die Oberklasse bzw. das Interface dann wiederum implementieren.

Auf diese Weise lassen sich weitere Klassen, welche das gleiche Interface implementieren, schnell hinzufügen oder gegeneinander austauschen. Am Beispiel des AppointmentService wird dies erläutert:

1. Die Main-Methode des „SimpleCRMServers“ definiert einen neuen „contactService“ durch Aufruf der Methode getContactService() in der „ContactServiceFactory“
2. Die „ContactServiceFactory“ erzeugt ein neues Objekt der Klasse „ContactServiceImpl“
3. Das Objekt “ContactServiceImpl” implementiert wiederum das Interface „ContactService“
4. Das neue Objekt wird über die “ContactServiceFactory” nun wieder zurückgereicht an die Main-Methode um es dem dort definierten „contactService“ zuzuordnen.

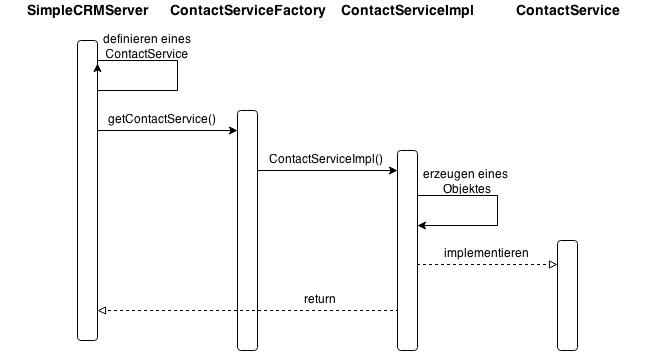


Abbildung 10 – Sequenzdiagramm Factory Pattern am Beispiel ContactService

# UML-Klassendiagramme der Anwendung

## Client

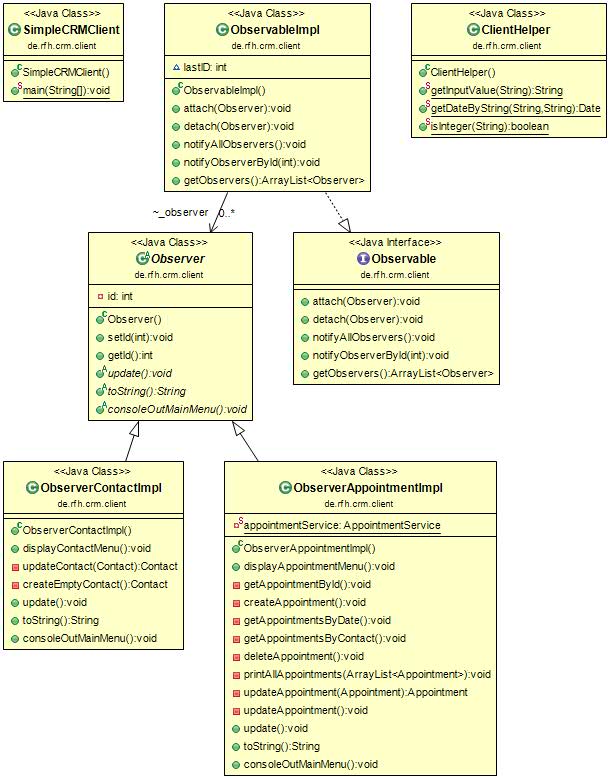


Abbildung 11 – UML-Klassendiagramm Client

## ContactService

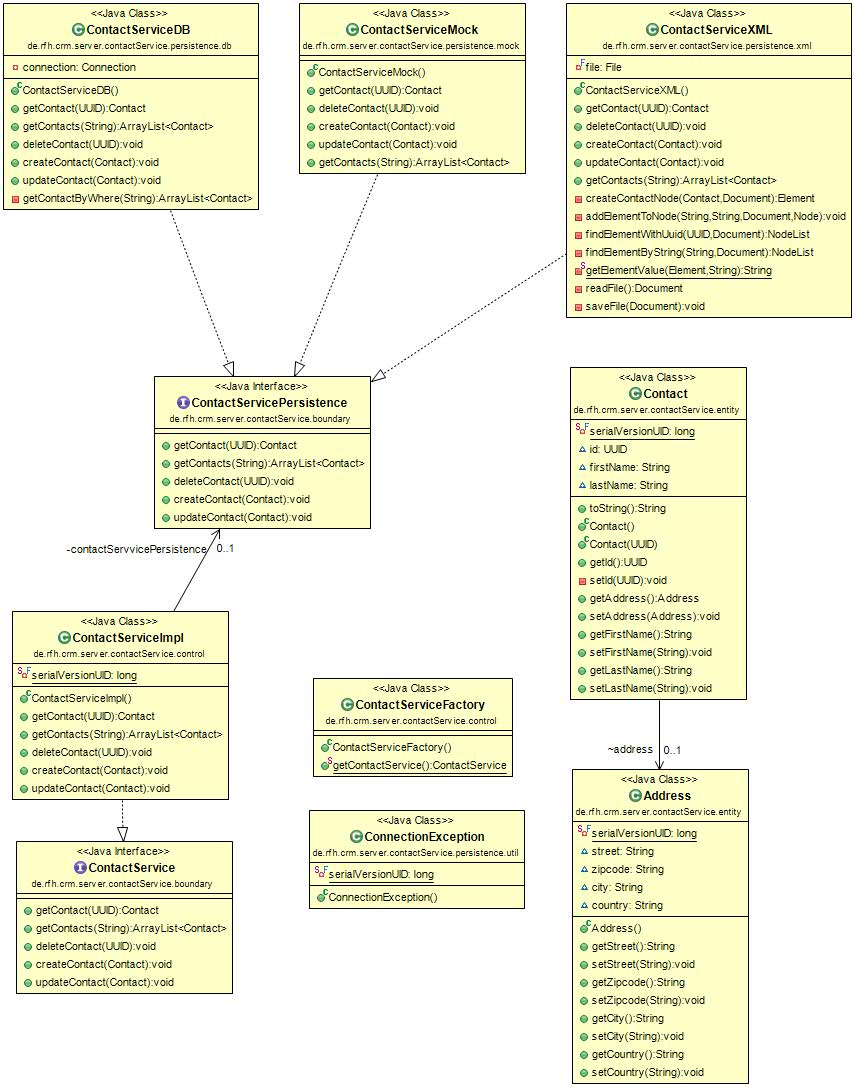


Abbildung 12 – UML-Klassendiagramm ContactService

## AppointmentService

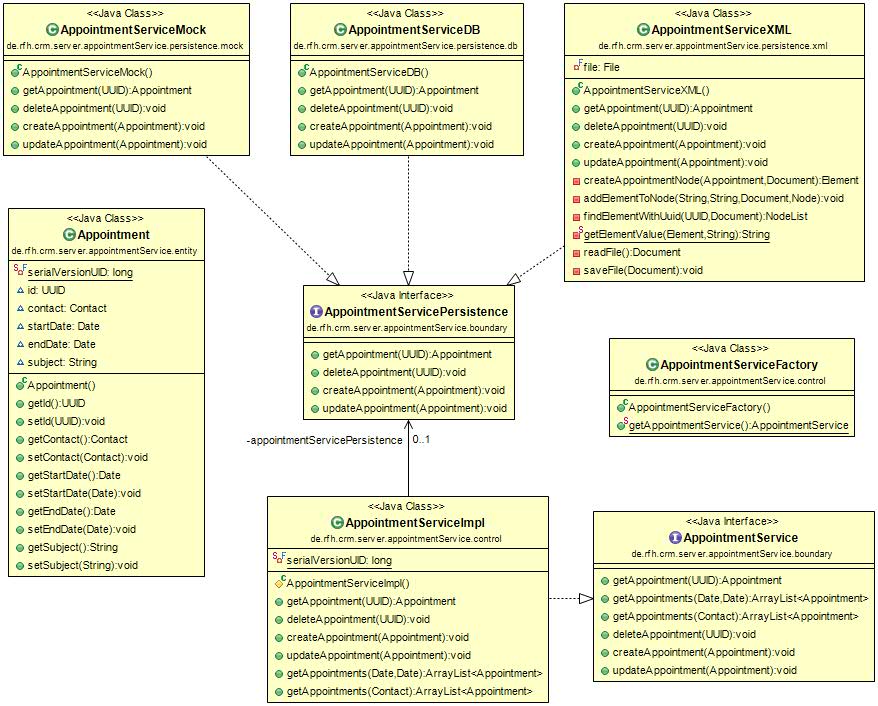


Abbildung 13 – UML-Klassendiagramm AppointmentService

## ConnectionManager

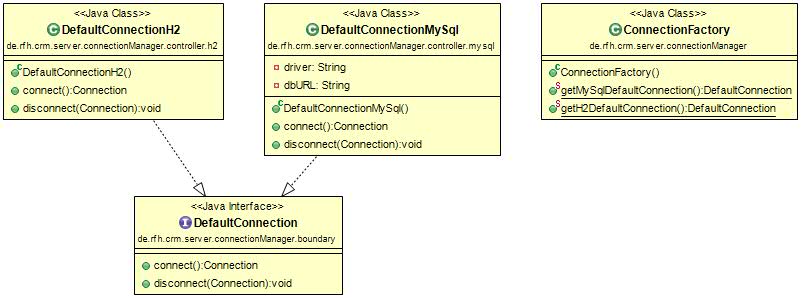


Abbildung 14 – UML-Klassendiagramm ConnectionManager

## Gesamt

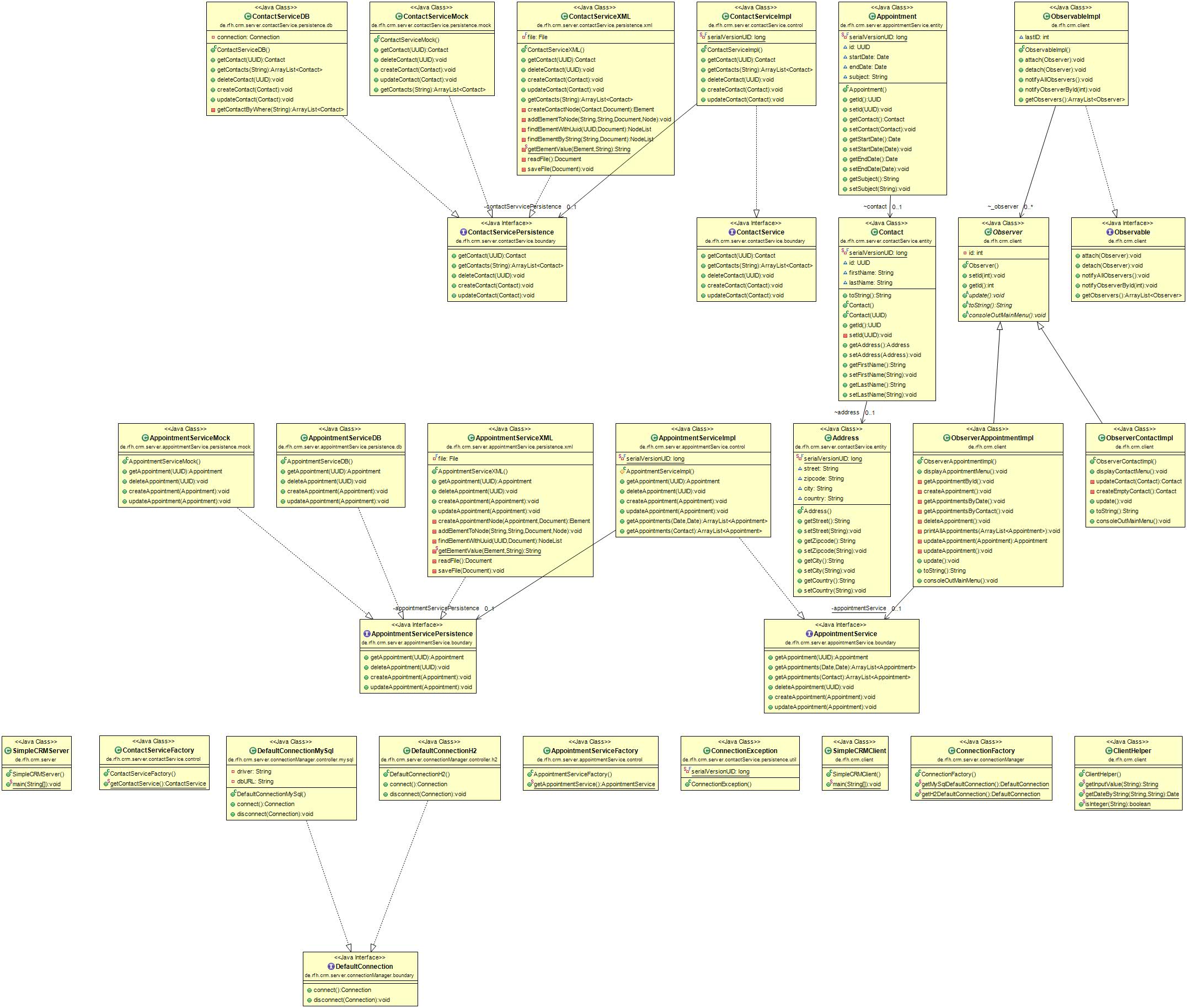
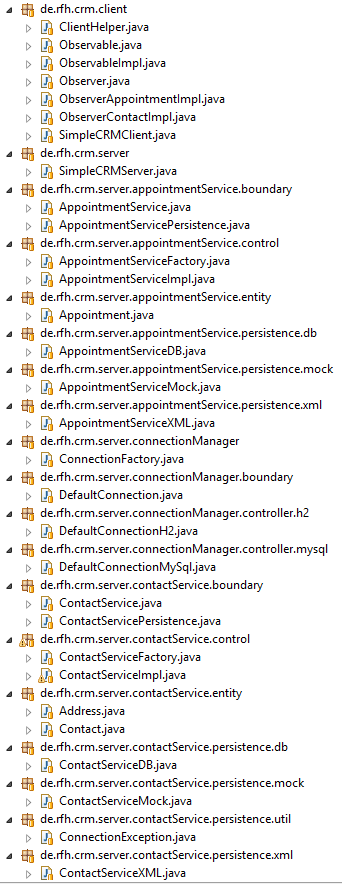


Abbildung 15 – UML-Klassendiagramm Gesamtübersicht

# Eingereichte Ergebnisse

Neben der Anwendung, welche in Dateiform und offener Java-Klassen einzelner Packages (siehe Abbilundg rechts) vorliegt, wurden diese Dokumentation, sowie eine Präsentation der Ergebnisse eingereicht.

Alle, für die Anwendung relevanten Daten wurden innerhalb des Unterverzeichnisses „code“ abgelegt. Dieses Verzeichnis enthält die binären Klassendateien (Verzeichnis „bin“), sowie verwendete Bibliotheken (Verzeichnis „lib“), den Java-Sourcode (Verzeichnis „src“) und die XML-Ausgabedateien (Verzeichnis „data“).

Das Unterverzeichnis „doc“ enthält die hier vorliegende Dokumentation.

Im Verzeichnis „praesi“ ist die Powerpoint-Datei abgelegt, welche während der Ergebnispräsentation verwendet wurde.

Das Verzeichnis „sql“ enthält ein SQL-Script, welches es ermöglicht die verwendete Datenbankstruktur in einen mySQL-Server zu importieren.

# Installationsanleitung

**Schritt 1**:   
Importieren Sie das Java-Projekt in eine Java Entwicklungsumgebung (Beispielsweise eclipse)

**Schritt 2**:   
Erstellen Sie eine neue Datenbank mittels des im Verzeichnis „sql“ abgelegten Datenbankscripts

**Schritt 3:**   
Passen Sie den ConnectionString (Attribut „dbURL“) in der Klasse „DefaultConnectionMySql“ an.

**Schritt 4:**   
Starten Sie die Serveranwendung

**Schritt 5:**   
Starten Sie die Clientanwendung und folgen Sie den Menuanweisungen.

# Fazit

Ziel war die Entwicklung eines Moduls, welche einen Use Case eines komplexen verteilten Java-Anwendung, abbildet. Zu Beginn wurde Im Workshop am 20.12.2014, wie in Punkt 3 beschrieben, das Thema Customer Relationship Management-System gewählt und die damit verbundenen Use Cases definiert. Hier wurden zunächst eine ganze Reihe von Use Cases benannt. Letztlich wurde sich auf spezielle damit verbundene Funktionen beschränkt.

Die Anforderung beschrieb jedoch nur die Umsetzung eines Use Cases. Gemeinsam wurde entschieden, die anderen Use Cases ebenfalls, zumindest in der Architektur, zu berücksichtigen und es 2 Services zu implementiert. Da sich die Funktionen der beiden Services größten Teils überschneiden (jeder Service beinhaltet die Funktionalitäten löschen, anlegen, manipulieren und lesen), ist es nicht wunderlich, dass sich der Aufbau und die darin enthaltene Struktur der Services unwesentlich unterscheiden. Lediglich bei der Speicherung von Daten unterscheidet sich der ContactService, auf Grund der Behandlung von zwei Entitäten, vom AppointmentService.

Bei der nachträglichen Betrachtung des Projekts, sind neben dem erfolgreichen Ergebnis, jedoch einige Punkte aufgefallen, welche in zukünftigen Projekten verbessert werden können.

Dies betrifft aus organisatorischer Sicht, die strukturierte Aufteilung von Aufgaben und die damit verbundene Verantwortung für den Teilbereich. Auch Zwischenzeitlich bietet es sich an, noch öfter als bereits durchgeführt, via Videokonferenzen den aktuellen Stand zu teilen und sich gegenseitig bei Problemen zu unterstützen. Auf Grund der recht langen Vorlesungsreihe und dem damit verbundenen kurzen Umsetzungszeitraums konnten zwar einige Themen ausprobiert werden (wie das verwenden des Observer Patterns), jedoch fehlte die Zeit um weitere Techniken einzubringen, die den Code attraktiver machen.

Als Verbesserung aus technischer Sicht, ist unter anderem die Struktur der Datenbank zu erwähnen. Diese unterscheidet sich von der Logik der Java-Anwendung. So werden innerhalb der Java-Anwendung die Adressdaten von den Kontaktdaten getrennt, in der Datenbankstruktur jedoch nicht. Auch innerhalb der XML-Speicherung gibt es Verbesserungspotenzial. Hier sind die Daten nicht normalisiert. Hat ein Kontakt mehr als einen Termin, so werden die Informationen zum Kontakt mehrfach abgelegt. Um eine neue Technik zu testen, wurde Observer Pattern implementiert und zweckentfremdet für den Aufbau eines Menus.

Alles in Allem, konnte das Projekt entsprechend der Anforderungen fristgerecht fertiggestellt werden. Die Arbeit im Team hat allen Teilnehmern die Herausforderung der Programmierung in einer Gruppe gezeigt und somit die Erfahrung ehöht und weiteres Interesse geweckt.

Darstellungsverzeichnis

[Abbildung 1 - Use Case Diagramm 3](#_Toc410322481)

[Abbildung 2 - UML-Klassendiagramm der Entitäten 4](#_Toc410322482)

[Abbildung 3 – Server-Client-Architektur 5](#_Toc410322483)

[Abbildung 4 – konzeptioneller Modulaufbau Server-Client 5](#_Toc410322484)

[Abbildung 5 – Aufbau der Persistenzschicht 6](#_Toc410322485)

[Abbildung 6 - Datenbankmodell 6](#_Toc410322486)

[Abbildung 7 – Clientmenu 1 7](#_Toc410322487)

[Abbildung 8 – Clientmenu 2 8](#_Toc410322488)

[Abbildung 9 – UML Observer Pattern 8](#_Toc410322489)

[Abbildung 10 – Sequenzdiagramm Factory Pattern am Beispiel ContactService 9](#_Toc410322490)

[Abbildung 11 – UML-Klassendiagramm Client 10](#_Toc410322491)

[Abbildung 12 – UML-Klassendiagramm ContactService 11](#_Toc410322492)

[Abbildung 13 – UML-Klassendiagramm AppointmentService 12](#_Toc410322493)

[Abbildung 14 – UML-Klassendiagramm ConnectionManager 13](#_Toc410322494)

[Abbildung 15 – UML-Klassendiagramm Gesamtübersicht 13](#_Toc410322495)