Konzeption eines Mail Service

Ing. Thomas Herzog

BACHELORARBEIT

Nr. XXXXXXXXXX-B

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2016

Diese Arbeit entstand im Rahmen des Gegenstands

Software Engineering

im

Sommersemester 2016

Betreuer:

FH-Prof. DI Dr. Heinz Dobler

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 1. Februar 2016

Ing. Thomas Herzog

Inhaltsverzeichnis

Vorwort

Folgende Arbeit beschäftigt sich mit der Re-Implementierung einer Mail Lösung, welche verwendet wird um Mails, die innerhalb einer Applikation generiert oder angefordert werden, zu verschicken. Da neue Anforderungen definiert wurden, die sich mit der existierenden Lösung nicht mehr realisieren lassen wurde beschlossen dass diese Lösung reimplementiert werden soll. Hierbei wird diese Arbeit den Fokus auf die existierende wie auch mögliche Implementierung bzw. deren Design legen. Die existierende Lösung weißt ein Alter von 10 Jahren auf, wurde mehrmals erweitert und ist auch dementsprechend designt.

Diese Maillösung wird von der Firma curecomp GembH verwendet, die eine Cloud basierte SRM-Lösung betreibt (Supplier-RelationShip-Management), deren Prozesse den Versand von E-Mails erfordern wobei diese E-Mails einerseits in verschiedene Bereiche zu unterteilen sind wie z.B.: Systemmails, Lieferverzugsmeldungen, Gutschriften, Bestellbestätigungen usw. und anderseits auch von verschiedenen System der SRM-Lösung erzeugt werden.
Bei diesen System handelt es sich einerseits um ein Datenimport/-export System (IIB - IBM-Integration-Bus) und anderseits um eine Web Applikation (Websphere - JSF). Beide Systeme sollen in der Lage sein einfach E-Mail Nachrichten eines bestimmten Typs zu versenden, wobei eine Nachricht dynamisch mit Daten versorgt werden soll und auf Basis einer Vorlage die E-Mail erstellt werden soll. Die Anhänge sollen ebenfalls dynamisch von verschiedenen Ressourcen geladen werden können. Eine versendete E-Mail soll über einen bestimmten Zeitraum hinweg weidervesendbar sein und ebenso sollen Log Daten der Metadaten vorhanden sein, die es erlauben den Lebenszyklus einer E-Mail nachträglich nachzuvollziehen. Z.B.: Wann wurde eine E-Mail and wen wie oft von wem versendet?

Beide Systeme sollen mit dem Mail Service System in derselben Art und Weise kommunizieren und interagieren, wobei aber die Systeme eine volle Integration der Client API zur Verfügung gestellt werden soll. Also sollen alle System spezifischen Ressourcen von der Client API nutzbar sein, wodurch ein gewisser Grad der Abstraktion der Implementierung zu gewährleisten ist.

// TODO: Kurzbeschreibung der Systemarchitektur und Komponenten (halbe Seite) // TODO: Zeichnung der System für die Visualisierung mit existierenden Mailservice.

Kurzfassung

// TODO: Zusammenfassung nach Abschluss der Arbeit in Deutsch

Abstract

// TODO: Add sumarry after thesis has been finished in english

Kapitel 1

Einleitung

1.1 Zielsetzung

Diese Bachelorarbeit dient als Vorarbeit für die Re-Implementierung einer existierenden Maillösung der Frima curecomp GembH, wobei in dieser Arbeit der Fokus auf die Re-Implementierung des Designs gelegt wird. Diese Arbeit wird aufzeigen wie das existierende System in seinen Bereichen wie Client-API, Datenbank Integration und das Erstellen der E-Mails, in weitere Folge 'E-Mail Vorlagen' genannt, zurzeit funktioniert und designt wurde. Dieses existierende Design wird analysiert und ein Verbesserungsvorschlag eingebracht wie ein mögliches neues Design aussehen könnte. Die Konzepte der Implementierung wie die verwendeten Software Muster, die Grundlage der Designentscheidung sowie deren Vorteile und Nachteile werden in dieser Arbeit diskutiert.

Da meiner Meinung nach die existierende Implementierung einige Mängel und Designfehlentscheidungen beinhaltet und dadurch weder flexibel noch erweiterbar ist, wurde der Fokus dieser Arbeit auf das Design dieser Maillösung und dessen Verbesserung gesetzt, wobei diese Arbeit wiederum als Grundlage für die praktische Bachelorarbeit dienen soll, die die Umsetzung der hier entwickelten und diskutierten Konzepte beinhalten soll.

Als Unterstützung wurden folgende beiden literarischen Werke gewählt:

1. Refactoring to patterns[1]
2. Refactoring Databases[2]

[1] Author: Joshua Kerhievsky, ISBN: 0-321-21335-1

[2] Author: Scott W.Ambler/Pramod J.Sadalage, ISBN: 0-321-29353-3

1.2 Anforderungen

Über die Zeit haben sich die Anforderungen an das Mailsystem derartig geändert, dass diese nicht mehr mit der existierenden Implementierung umgesetzt werden können. Dies liegt vor allem daran dass die existierende Implementierung aus dem Jahre 2002 stammt und man dessen umgesetztes Design mittlerweile vollständig ausgereizt hat. Andererseits muss man aber auch beachten dass die angeschlossenen Systeme sich über die Zeit evolutioniert, also weiterentwickelt, haben und daher auch neue Anforderungen stellen. Ebenso haben sich auch die Kundenanforderungen geändert.
Aufgrund dieser Ausgangssituation wurden folgende technische Anforderungen definiert, die im folgenden aufgelistet sind.

- Persistenz der versendeten E-Mails inklusive Anhänge.
- Integration in andere Systeme in Form von Rest-API (Rest-Service)
- Metadaten sollen für E-Mails definiert werden können (z.B.: Gruppierung)
- Konfigurierbarkeit des Mailversands, Speicherung, Archivierung, ...

Das Gesamtsystem soll hierbei so flexibel wie möglich sein und muss in der Lage sein zumindest mit dem Systemen der Firma curecomp zu interagieren.

1.3 Überblick Ist-System

Der jetzige Mailservice besteht im Grunde nur aus einer Konsolen Applikation, einigen Datenbank Tabellen, und den Implementierungen der DAO *(Data-Access-Object)* der beiden angeschlossenen Systeme. Mann sieht einerseits dass hier die Redundanz in Form der DAO Implementierungen der angeschlossenen Systeme Inkonsistenz mit sich bringt und andererseits dass die Koppelung über die Datenbank unflexibel ist und alle angeschlossenen Systeme mit ihren eigenen Implementierungen darauf zugreifen und es kein übergeordnetes System gibt, welches die allgemeinen Aktionen weg abstrahiert, bündelt und allen Systemen zur Verfügung stellt.

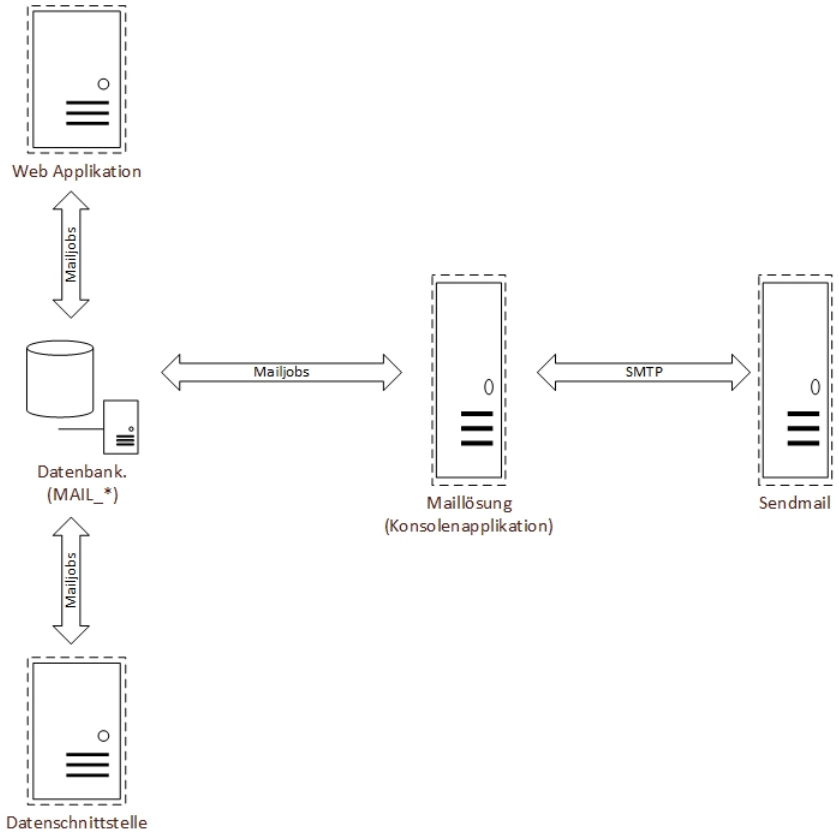

Abbildung 1.1: Diese Abbildung zeigt den konzeptionellen Aufbau der existierenden Maillösung und den jetzigen Kommunikationsweg zwischen den Systemteilnehmern

Kapitel 2

Die Konsolen Applikation

Da in dieser Arbeit das Design der bestehenden Implementierung diskutiert und ein Designvorschlag eingebracht werden soll, muss man zuerst die Struktur und das Design der bestehenden Implementierung verstehen lernen. Da sich die meiste Geschäftslogik in der Konsolen Applikation befindet wird im folgenden dessen Design und Struktur diskutiert.

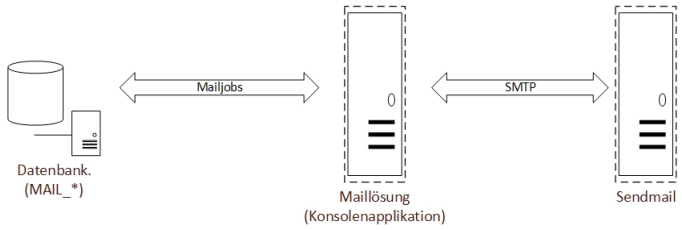

Abbildung 2.1: Teilsystem Konsolen Applikation und Sendmail

2.1 Klassenhierarchie

Um die Struktur der Konsolen Applikation zu verstehen wenden wir uns den implementierten Klassenhierarchien der einzelnen Softwarekomponenten wie E-Mail Typen und DAOs (Datenzugriffsobjekte) zu. Diese beiden Softwarekomponenten stellen den Hauptteil der Software dar, daher beginnen wir mit der Analyse dieser Komponenten.

2.1.1 E-Mail Typen

Einleitend betrachten wir die implementierte Klassenhierarchie der E-Mail Typen mit einem Ausschnitt aus der Klassenhierarchie. Dieser Ausschnitt illustriert sehr gut die vorhandene Struktur der Klassenhierarchie.

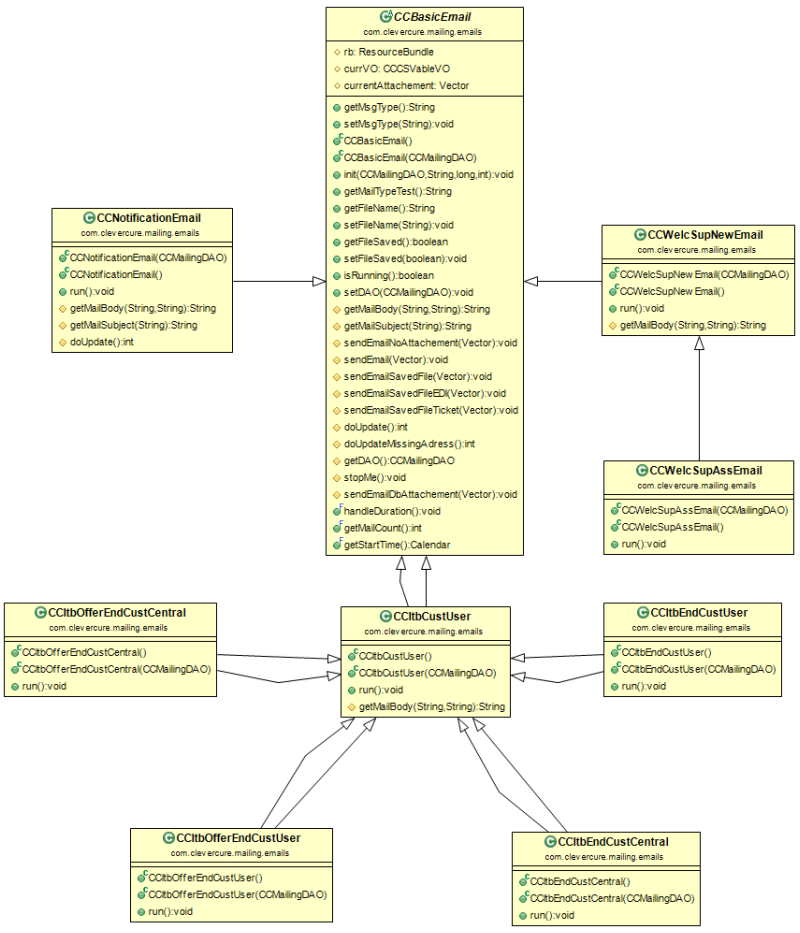

Abbildung 2.2: Die Basisklasse *BasicEmail* mit einigen Implementierungen

Man hat sich hier für das *Factory Method Muster* entschieden wobei die Basisklasse *BasicEmail* eingeführt wurde, welche die gesamte Logik für den Aufbau und Versand einer E-Mail in sich kapselt und die einzelnen konkreten Implementierungen, in Form von abgeleiteten Klassen, die die einzelnen E-Mail Typen darstellen. Wie im Diagramm bei der Klasse *CCItbCustUser* ersichtlich wurden auch E-Mail Typen implementiert, die ihrerseits wieder als Basisklasse dienen für untergeordnete E-Mail Typen. Aufgrund des vergebenen Namens kann man annehmen dass es sich hier um E-Mail Benachrichtigungen für einen Benutzer eines Kunden des Moduls ITB handelt, die man versucht hat zu gruppieren.
Die Gruppierung wurde hier durch Einführung einer eigenen Vererbungshierarchie auf Basis einer eingeführten Basisklasse *CCItbCustUser*, die ihrerseits die eigentliche Basisklasse *BasicEmail* weg abstrahiert, realisiert.

An sich ist dies kein schlechter Ansatz jedoch erwarte ich mir von einer konkreten Implementierung, dass diese auch ein gewisses Maß an Logik beinhaltet, die das Factory-Method Muster rechtfertigt, jedoch ist dies in den meisten Fällen nicht der Fall. Solche Klassenhierarchien einzuführen nur um verschiedene Datentypen zu erhalten, welche einen E-Mail Typ definieren, welcher nicht einmal in den angeschlossenen Systemen verwendet wird sondern nur innerhalb dieser Konsolen Applikation finde ich übertrieben. Es gibt weitaus einfachere, weniger aufwendige und Code produzierende Ansätze mit denen E-Mail Typen abgebildet werden können.

2.1.2 Datenzugriffsobjekt(e)

Nachdem wir die Klassenhierarchie der E-Mail Typen analysiert haben wenden wir uns nun der Datenbankzugriffsschicht zu, die mit einem einzigen DAO (Datenzugriffsobjekt) implementiert wurde.

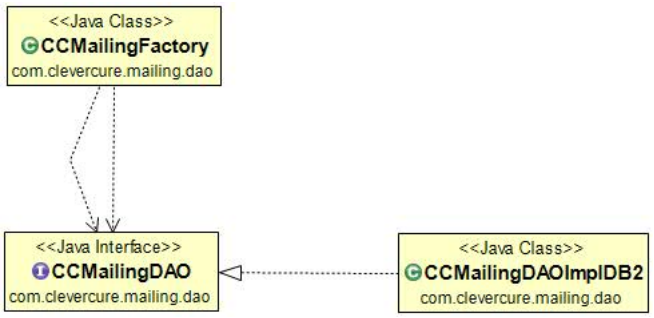

Abbildung 2.3: Die *CCMailingDAOImplDB2* Implementierung mit seinem Interface *CCMailingDAO* mit seiner Factory *CCMailingFactory*

Obwohl man eine Klassenhierarchie für die E-Mail Typen eingeführt hat, hat man dies bei den DAOs (Datenzugriffsobjekte) nicht getan. Es gibt also eine einzige DAO Implementierung, die alle Datenbankabfragen über alle E-Mail Typen hinweg beinhaltet und diese über eindeutige Methodennamen identifiziert. Des Weiteren wurde zwar eine Factory für dieses DAO eingeführt, aber eine Factory für ein DAO mit *Class.forName("*.DaoImplDb2")* erscheint fast sinnlos, obwohl man hier noch die Begründung finden könnte, dass diese Factory eingeführt wurde um zumindest die konkrete Datenbank zu abstrahieren und die Möglichkeit zu schaffen die DAO Implementierung für DB2 auf z.B.: Oracle zu ändern. Man würde hier erwarten das die DAO Implementierungen kontextabhängig - also auf Modulebene - implementiert worden wären und dass die Factory dynamisch mit der zu verwendenden Implementierung initialisiert werden kann, also parametriebar ist. Dann wäre es aber auch erforderlich die Datenbank spezifischen Implementierungen über ein eigenes Artifakt zur Verfügung zu stellen und nicht das alle Ressourcen sich innerhalb ein und desselben Artifaktes befinden, also die Interface Spezifikationen und dessen Implementierungen.

2.2 Implementierung

Da wir nun die Klassenhierarchien kennen gelernt haben wenden wir uns deren Implementierung zu. Im folgenden werden die Implementierungen der Klasse *CCItbCustUser* und dessen Ableitungen diskutiert. Im Punkt 2.1 wurde behauptet dass diese Ableitungen eingeführt wurden um E-Mail Typen zu gruppieren. Man könnte aber auch davon ausgehen dass diese eigene Vererbungshierarchie eingeführt wurde um Funktionalitäten für die einzelnen E-Mails zu kapseln. Analysieren wir nun diese Implementierungen um zu sehen welche Funktionalitäten in einen E-Mail Typ implementiert wurden.

Die folgende Implementierung dient als Beispiel für eine Implementierung eines E-Mail Typs.

Programm 2.1: CCItbCustUser E-Mail Typ Implementierung

```
1 public class CCItbCustUser extends CCBasicEmail {
2
3   private Map cache = new HashMap();
4
5   public CCItbCustUser() {
6     super();
7   }
8
9   public CCItbCustUser(CCMailingDAO dao) {
10    super(dao);
11  }
12
13  @Override
14  String getMailType() {
15    return "ISCU";
16  }
17
18  @Override
19  public void run() {
20    try {
21      sendEmailNoAttachement(getDAO().getItbStartCustUserMailText());
22    } catch (DAOSysException ex) {
23      LOG.error("DAOSysException in CCItbCustUser.run: ",
24            ex);
25    } finally {
26      stopMe();
27    }
28  }
29
30  @Override
31  protected String getMailBody(String bodyKey, String bodySQLKey)
32    throws DAOSysException {
33    int lanId = ((CCItbVO)currVO).getLanguageId();
34    int itbhId = ((CCItbVO)currVO).getItbhID();
35    String body = "";
36    String key = itbhId + "_" + lanId;
37    if (cache.containsKey(key)) {
38      body = (String) cache.get(key);
39      LOG.debug("48: Got from cache key: " + key
40                + " body: " + body);
41    } else {
42      Object [] allParams = getDAO().getItbCustData((CCItbVO)currVO, 19)
      ;
43      // Message body parameters retrieved from result set for body template
44      MessageFormat form = new MessageFormat(rb.getString(bodyKey).trim
      ());
45      body = form.format(params);
46      cache.put(key, body);
47      LOG.debug("48: DB access for the key: " + key
48              + " got body: " + body);
49    }
50    return body;
51  }
52 }
```

In dieser Implementierung ist gut zu erkennen, dass die einzelnen E-Mail Typen - oder mit anderen Worten - die einzelnen Implementierungen der Basisklasse *BasicEmail* lediglich folgende Funktionalitäten implementieren:

- *getMailType()*
  Bereitstellen eines eindeutigen Schlüssels, der diesen E-Mail Typ identifiziert.
- *getMailBody()*
  Erstellen der E-Mail Nachricht aus einer Vorlage, welche mit Parametern - Daten werden aus Datenbank gelesen - befüllt wird.
- *run()*
  Einstiegspunkt des E-Mail Typs, wo definiert wird welche Art von E-Mail erstellt werden soll. (*BasicEmail* stellt mehrere Methoden zur Verfügung)

Kapitel 3

Client Implementierung

Da die Kommunikation ausschließlich über die Datenbank erfolgt ist eine echte Client API nicht vorhanden. Alle Systeme implementieren das Erstellen der Mailjob Einträge selbstständig und es wird keine gemeinsame Client API verwendet, was einen gewissen Grad der Inkonsistenz bei Änderungen der Datenstruktur sowie der Semantik der Daten mit sich bringt. Als Beispiel wird hier die Semantik der einzelnen Spalten angeführt, die zwar technisch sich innerhalb einer Datentyp Domain befinden, sich die semantische Bedeutung der enthaltenen Daten aber unterscheidet. Dies ist zwar Teil der Datenbank Integration aber die Tatsache dass diese Semantik rein innerhalb der Applikation abgebildet werden kann und nicht über die Datenbankfunktionalitäten wie z.B.: Fremdschlüssel wird dieser Teil hier diskutiert. Da die Maillösung über SQL Abfragen die Daten für die E-Mail Nachricht erhält müssen auch Parameter bereitgestellt werden, die in der SQL Abfrage gesetzt werden. Diese Parameter wurden *'generisch'* über eine festgesetzte Anzahl von Tabellenspalten abgebildet, dessen enthaltene Daten je nach E-Mailtyp anders interpretiert werden. Es kann also sein das ein Eintrag einer Tabellen für die Spalte *COL_1* einen String enthält mit dem Wert *'Thomas Herzog'* und ein anderer Eintrag einen string mit dem Wert *'14'* der in der SQL Abfrage aber als Integer Datentyp behandelt wird. Die Systeme die Mailjobs erzeugen müssen sich also dem Kontext einer E-Mail bewusst sein, sowie müssen berücksichtigen, dass die richten Spalten der Tabelle *MAIL_JOB* mit den richtigen Daten befüllt werden, wobei auch beachtet werden muss in welchen Datentyp der Datensatz in weiterer Folge durch die Mailösung interpretiert wird.
Dies ist in meinen Augen sehr inkonsistent und fehleranfällig und hat auch schon einige Probleme verursacht.

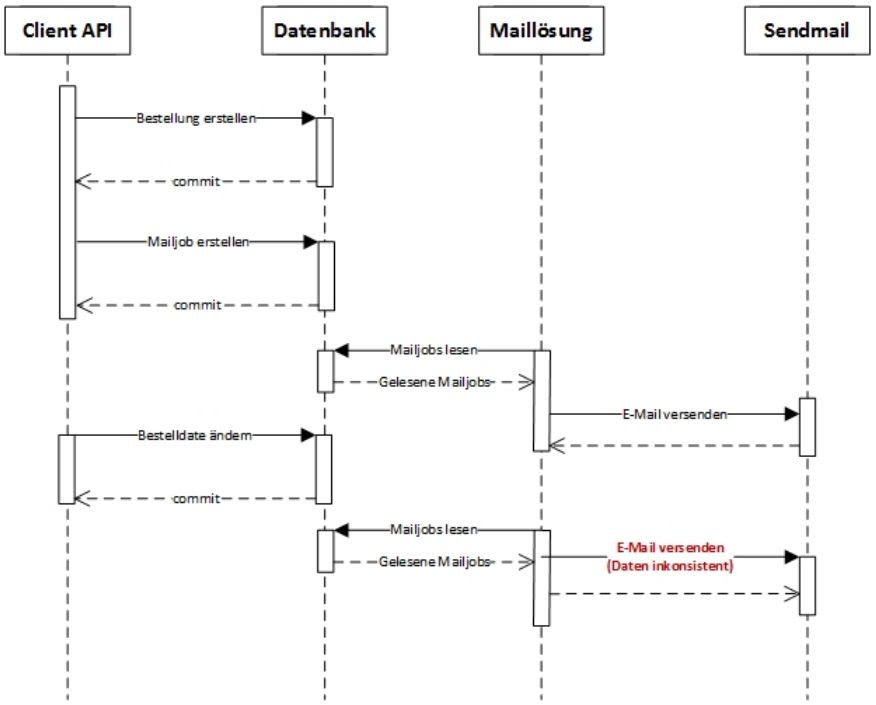

Abbildung 3.1: Diese Abbildung zeigt das Problem der inkonsistenten Daten beim erneuten E-Mailversand einer bereits versendeten E-Mail, mit dem Beispiel einer angelegten und anschließend geänderten Bestellung, auf

3.0.1 Datenbank Integration

3.0.2 E-Mail Vorlagen

3.1 Soll-System

3.1.1 Client-API

3.1.2 Datenbank Integration

3.1.3 E-Mail Vorlagen

3.2 Arbeiten in Englisch

Quellenverzeichnis