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 Konzeption einer Mail-Anwendung für die Firma curecomp GembH, welche in weiterer Folge als *CleverMail* bezeichnet wird, die eine bestehende Mail-Anwendung, in weiterer Folge *CCMail* genannt, ablösen soll. Die Firma curecomp GembH ist ein Dienstleister im Supplier Relationship Management (SRM) und betreibt eine Softwarelösung, die aus zwei Anwendungen besteht:

1. *CleverWeb*
   Eine Web-Anwendung für den webbasierten Zugriff
2. *CleverInterface*
   Eine Schnittstellen-Anwendung für die Anbindung der ERP-Systeme der Kunden und Lieferanten

Beide Anwendungen erfordern den Versand von E-Mail-Nachrichten um verschiedene Systemzustände den Benutzern mitzuteilen wie z.B.:

- Fehlermeldungen
- Statusänderungen bei Bestellungen (erstellt, geliefert, storniert, ...)
- Lieferverzugsmeldungen
- ...

Es wurden neue Anforderungen für *CCMail* definiert, die sich nicht mehr in *CCMail* integrieren lassen. Dies ist begründet in dem Design und der Implementierung von *CCMail*.

Diese Arbeit wird sich einerseits mit der Diskussion des bestehenden Designs und der bestehenden Implementierung von *CCMail* befassen und andererseits ein Konzept für *CleverMail* einbringen. Das eingebrachte Konzept soll als Basis für die praktische Bachelorarbeit dienen, in der das eingebrachte Konzept umgesetzt werden soll.

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

Das Ziel dieser Arbeit liegt in der Konzeption von *CleverMail*, welche die bestehende Anwendung *CCMail* ablösen soll. Bevor dieses Konzept erstellt wird, wird die bestehende Anwendung *CCMail* insbesondere dessen Design und Implementierung diskutiert damit aufgezeigt werden kann welche Designentscheidungen und Implementierungen ein Erweitern von *CCMail* verhindern. Im neuen Konzept sollen die in *CCMail* gemachten Fehler und Fehlentscheidungen berücksichtigt werden damit *CleverMail* auch zukünftig neuen Anforderungen gewachsen ist und sich neue Anforderungen ohne größere Probleme integrieren lassen. Zukünftige Anforderungen sind zwar schwer vorauszusagen jedoch kann man sich bei seinen Designentscheidungen, der Wahl der verwendeten Softwaremuster und Anwendungsarchitektur auf neue Anforderungen bzw. Änderungen an der bestehenden Anwendung gut vorbereiten.

Für die Konzipierung von *CleverMail* wurden folgende technischen Grundvoraussetzungen definiert:

1. Java 1.8
2. IBM-DB2 (Datenbank)

Das Konzept von *CleverMail* soll vor allem auf neue Konzepte und Frameworks in Java konzentrieren, da man hier keine Rücksicht auf Rückwärtskompatibilität nehmen muss.

Als Unterstützung für diese Arbeit wurden folgende beiden literarischen Werke gewählt:

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

Über die Zeit haben sich die Anforderungen an *CCMail* derartig geändert, dass diese nicht mehr in *CCMail* integriert werden können. Wie im Vorwort bereits erwähnt liegt dies vor allem an dem Design von *CCMail*. *CCMail* wurde im Jahre 2002 implementiert in *Java 1.4* implementiert und hatte daher nicht die technischen Möglichkeiten die uns heute zur Verfügung stehen. Als Erinnerung, Java Generics wurden erst mit der Version *Java 1.5 auch bekannt als Java 5.0* implementiert. Auch sind bis heute alle Erweiterungsmöglichkeiten in *CCMail* ausgeschöpft worden, wobei eine Erweiterung alleine schon wegen dem großen Versionsunterschied zwischen *Java 1.4* und *Java 1.8* sinnlos erscheint. Weiterentwicklungen fanden zwar in *CleverWeb* und *CleverInterface* statt jedoch scheint es so, dass *CCMail* hier vernachlässigt wurde, was dazu geführt hat, dass ein derartig großer technologischer Unterschied eingetreten ist.

[1] Author: Joshua Kerhievsky, ISBN: 0-321-21335-1
[2] Author: Scott W.Ambler/Pramod J.Sadalage, ISBN: 0-321-29353-3

Kapitel 2

Gesamtsystem

In diesem Kapitel wird der Systemaufbau und insbesondere die Integration von *CCMail* in dasselbe diskutiert. Folgendes Diagramm illustriert den Systemaufbau und die Integration von *CCMail* in das Gesamtsystem.

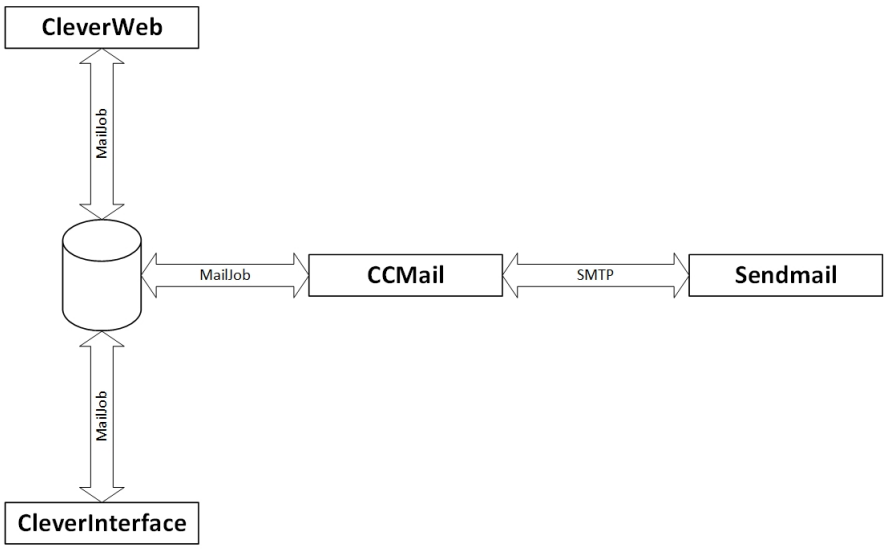

Abbildung 2.1: Systemaufbau und Integration von *CCMail*

Aus diesem Diagramm lässt sich ableiten dass das Kernstück des E-Mail Versandes die Datenbank ist. In der Datenbank werden die zu versendenden E-Mail-Nachrichten als sogenannte *MailJob* in der Datenbank gespeichert und anschließend von *CCMail* ausgelesen und verarbeitet. Ein *MailJob* ist hierbei als eine Entität einer Datenbanktabelle anzusehen.

CleverWeb und *CleverInterface* erstellen über ihre Datenbankzugriffsschicht

MailJob Entitäten in der Datenbank, welche zeitgesteuert von *CCMail* ausgelesen, verarbeitet und versendet werden. *CCMail* ist als eine Konsolen-Anwendung Implementiert und enthält alle Ressourcen, die es benötigt um die *MailJob* Entitäten zu verarbeiten.

2.1 Prozessübersicht

Im folgenden wird ein Beispiel für den Gesamtprozess diskutiert, also vom Anlegen eines *MailJob* bis hin zum Versand der eigentlichen E-Mail-Nachricht. Als Kernkomponente des E-Mail-Versandes wurde die Datenbank identifiziert, die die *MailJob* Entitäten hält. Dieser Ansatz ist an sich nicht als schlecht anzusehen, jedoch verbirgt sich hier eines der Hauptprobleme des E-Mail-Versandes. Nämlich die Inkonsistenz der versendeten E-Mail-Nachrichten, durch die Zeitdifferenz zwischen dem Anlegen eines *MailJob* und dem tatsächlichen Versand der E-Mail-Nachricht wie in folgendem Sequenz-Diagramm illustriert wird.

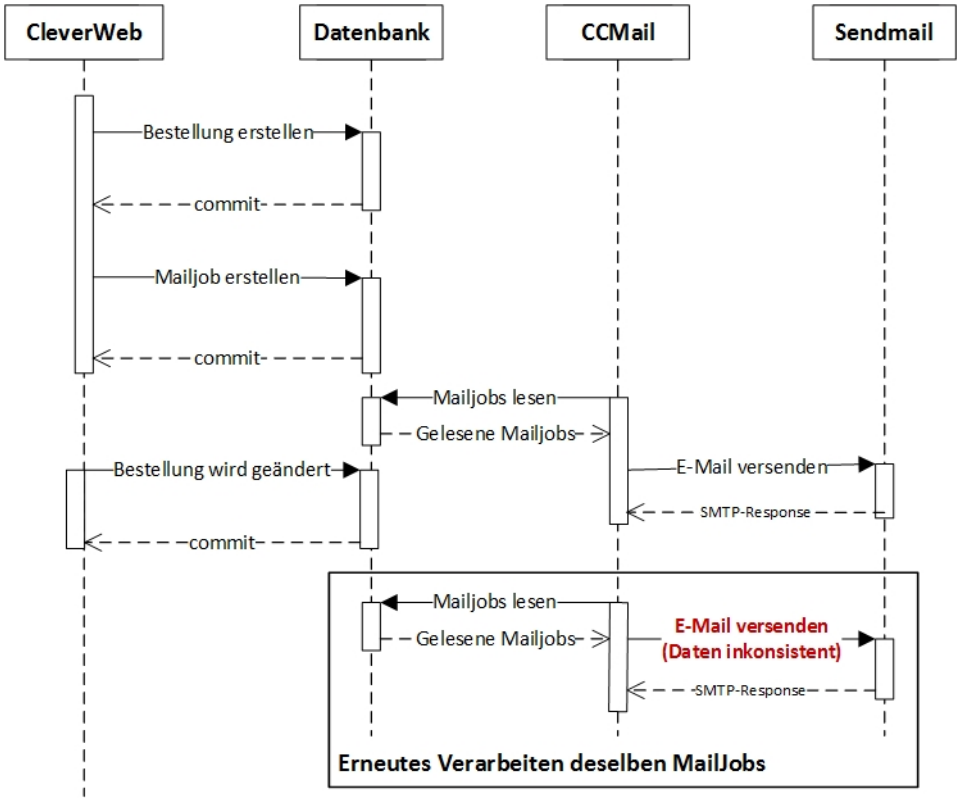

Abbildung 2.2: Beispiel für den Gesamtprozess eines E-Mail Versandes

Eines der Hauptprobleme an diesem Prozess ist die Inkonsistenz, was in der

Art und Weise wie die *MailJob* Entitäten verarbeitet werden begründet ist. Ein *MailJob* hält die Informationen für den Versand einer E-Mail wobei hierbei nicht die gesamte E-Mail-Nachricht gespeichert ist sondern lediglich die Parameter, die in einer Datenbankabfrage genutzt werden um die Daten für die E-Mail-Nachricht zu erhalten. Sollten sich die Datenbank Entitäten der involvierten Tabellen ändern, wie in diesem Beispiel eine Bestellung, so ist dieselbe E-Mail-Nachricht, die bereits versendet wurde nicht mehr wieder herstellbar, da sich die zugrundeliegenden Daten geändert haben.

Ein Weiteres Problem liegt in der zeitgesteuerten Abarbeitung der *MailJob* Entitäten von *CCMail*. Es wurde lange Zeit nicht geprüft ob bereits ein *CCMail* Prozess gestartet wurde bevor dieser gestartet wird. Dies hat dazu geführt dass es vorkam das mehrere Prozesse gleichzeitig die *MailJob* Entitäten verarbeiten und daher wurden E-Mail-Nachrichten mehrmals versendet.

Kapitel 3

CCMail

Nachdem das Gesamtsystem und ein Beispiel für den Prozess eines E-Mail-Versandes diskutiert wurden wird sich nun mit dem Design und der Implementierung von *CCMail* befasst. *CCMail* wurde als Konsolen-Anwendung implementiert und hält alle implementierten E-Mail-Typen, die zur Verfügung stehen wie:

1. E-Mail-Vorlagen
2. Datenbankabfragen
3. E-Mail-Typ spezifische Implementierungen

Es wurde hierbei eine Klasse namens *CCBasicEmail* implementiert, die alle nötigen Funktionalitäten für den Versand einer E-Mail-Nachricht enthält, sowie eine Klasse namens *CCMailingDao* die alle Datenbankabfragen über alle E-Mail-Typen hält.

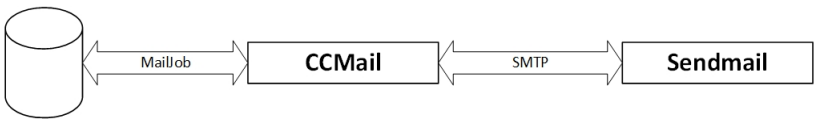

Abbildung 3.1: Teilsystem *CCMail*

3.1 Klassenhierarchien

Um das Design von *CCMail* zu illustrieren wird im Folgenden näher auf die Klassenhierarchien von *CCMail* eingegangen. Im Grunde besteht *CCMail* aus den beiden folgenden Softwarekomponenten:

1. *CCBasicEmail*
   Wurzelklasse aller E-Mail-Typen die als abgeleitete Klassen von *CCBasicEmail* implementiert wurden
2. *CCMailingDao*
   Schnittstelle zur Datenbank, welche alle Abfragen auf die Datenbank über alle E-Mail-Typen hinweg spezifiziert.

Die Klassenhierarchien geben einen guten Einblick in das Design und die Struktur von *CCMail* und werden aufzeigen was der Grundgedanke beim Design von *CCMail* war und warum man sich für das Design entschieden hat.

3.1.1 CCBasicEmail

Einleitend betrachten wir die Klassenhierarchie der Klasse *CCBasicEmail*, die die Wurzelklasse aller E-Mail-Typen darstellt.

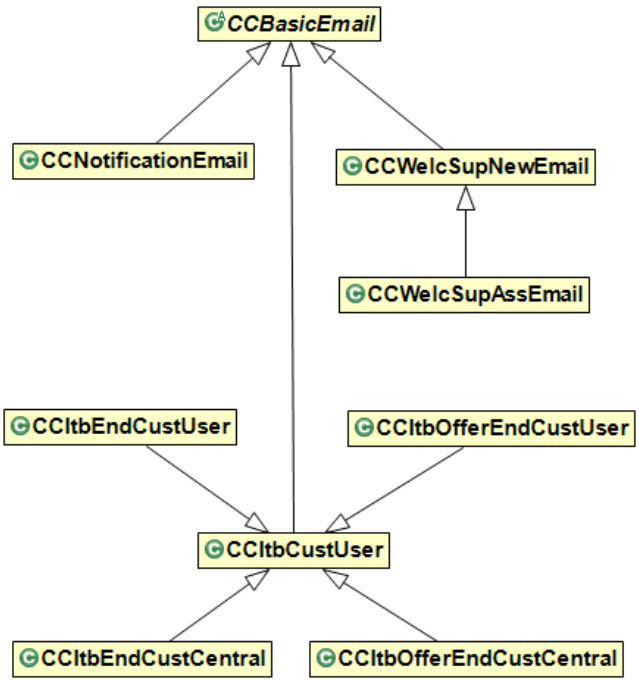

Abbildung 3.2: Auszug aus der Klassenhierarchie von *CCBasicEmail*

In diesem Klassendiagramm sieht erkennt man sehr gut dass man sich dazu entschieden hat die einzelnen E-Mail-Typen als eigene Klassen abzubilden. Somit ist jeder E-Mail-Typ auch als eigener Java Typ abgebildet. Am Beispiel der Klasse *CCItbCustUser* ist ersichtlich dass neben dem Abbilden eines E-Mail-Typs als eigene Java Klasse man ebenfalls eine eigene Subklassenhierarchie eingeführt hat um E-Mail-Typen, die in einem gemeinsamen Kontext sich befinden zu gruppieren. Ob dies ein guter Ansatz um Kontextabhängige Ressourcen gruppieren ist zu hinterfragen. Es gäbe hier andere Ansätze wie man eine solche Gruppierung hätte realisieren können, die flexible sind als eine Klassenhierarchie. Eine Klassenhierarchie ist an sich starr und nicht flexibel und Änderungen an der Struktur können sich negativ in der Gesamtstruktur auswirken.

3.1.2 CCMailingDao

Im Gegensatz zur Strukturierung der E-Mail-Typen hat man sich bei der Datenzugriffsschicht nicht dazu entschieden diese kontextabhängig zu gruppieren. Hier wurden alle Datenbankabfragen in einer einzigen Schnittstelle spezifiziert ohne Rücksichtnahme auf deren Kontext.

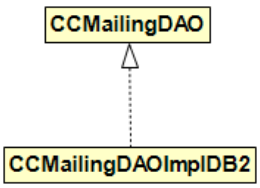

Abbildung 3.3: Klassenhierarchie von *CCMailingDao*

Diese Klassenhierarchie ist sehr einfach, da man sich nicht für eine Gruppierung entschieden hat. Hier ist zu bemängeln dass sich hier alle Datenbankabfragen über alle E-Mail-Typen hinweg befinden und man es versäumt hat hier Schnittstellen einzuführen, die die kontextabhängigen Datenbankabfragen spezifizieren. Mit einer Aufteilung auf mehrere Schnittstellen hätte man es sich vereinfacht diese die Datenbankabfragen zu warten. Mit diesem Ansatz ist man erstens gezwungen Präfixe für die Methodennamen einzuführen, da Namenskollisionen sehr wahrscheinlich sind und zweitens muss man darauf Acht geben bestehende Implementierungen bei einem Refaktorisieren einer oder mehrerer kontextabhängigen Implementierungen nicht zu verändern. Alle Implementierungen nutzen dieselben Ressourcen und müssen daher auf den kleinsten gemeinsamen Nenner zusammengeführt werden, oder man führt hier wiederum eigene Ressourcen ein, die sich durch ihren Namen unterscheiden.

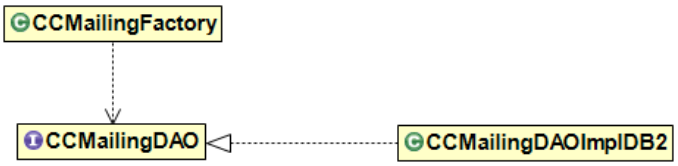

Abbildung 3.4: *CCMailingDaoFactory* für *CCMailingDao*

Zu kritisieren ist hier auch die Art und Weise wie eine Instanz von *CCMailingDao* erzeugt wird. Man nutzt hier den Factory-Pattern, jedoch wird hier statisch die zu verwendende Implementierung in der Factory *CCMailingDaoFactory* definiert was das Austauschen der Implementierung unmöglich macht. Zusätzlich befindet sich die Schnittstelle zusammen mit ihrer Implementierung in ein und demselben Artefakt. Dies ist als ein sehr halbherziger Versuch zu werten die Implementierung austauschbar zu machen.

3.2 Implementierung CCItbCustUser

Nachdem die Klassenhierarchien diskutiert wurden wird im folgenden die Implementierung der Klasse *CCItbCustUser* und dessen Ableitungen diskutiert. Diese Implementierung dient als Beispiel für die restlichen E-Mail-Typ Implementierungen, die nach dem selben Prinzip mit ähnlichen Umfang implementiert wurden. Im Punkt 3.1.1 wurde behauptet dass diese Ableitungen eingeführt wurden um E-Mail-Typen zu gruppieren. Man könnte aber auch davon ausgehen dass diese eigene Subklassenhierarchie eingeführt wurde um gemeinsame Funktionalitäten für die abgeleiteten E-Mail-Typen zu kapseln.
STOPPED HERE at 07.12.2015: 20:53

Programm 3.1: 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 4

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.

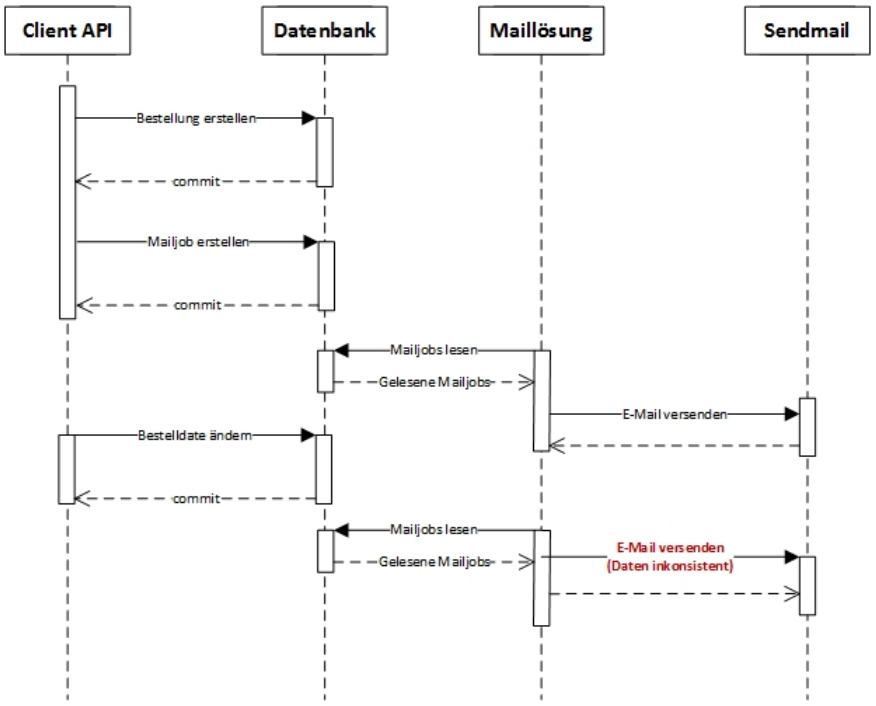

Abbildung 4.1: Diese Abbildung zeigt das Problem der inkonsistenten Daten beim erneuten E-Mailversand einer bereits versendeten E-Mail, mit dem Beispiel einer angelegten und anschließend geänderten Bestellung, auf

Quellenverzeichnis