Konzeption eines Mail-Service

Ing. Thomas Herzog

BACHELORARBEIT

Nr. S1310307011-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2016

Diese Arbeit entstand im Rahmen des Gegenstands

Software Engineering

im

Wintersemester 2015/16

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 29. Februar 2016

Ing. Thomas Herzog

Inhaltsverzeichnis

Kurzfassung

Die Analyse der bestehenden Anwendung *CCMail* wurden viele Fehlentscheidungen und Probleme ausgemacht, die dazu geführt haben, dass diese Anwendung neuen Anforderungen nicht mehr gerecht werden kann. Das Software-Designs sowie das implementierte Datenbankschema wurden zu lange nicht gewartet und haben sich zu lagen nicht den neuen Standards und Technologien angepasst, was dazu geführt hat, das ein Refaktorisieren einer Reimplementierung gleichzusetzen ist. Nichts desto trotz wurden im Konzept von *CleverMail* viele Eigenschaften von *CCMail* übernommen. Vor allem was den E-Mail-Versand selbst betrifft, der, bis auf die neuen Feature, beinahe glich verläuft. Im Konzept von *CleverMail* wurden vor allem neue Möglichkeiten, die mit der Verwendung der JEE-7-Plattform zur Verfügung stehen, vorgestellt. Vor allem das Problem, der Vorlagenparameter stellt eine Herausforderung dar, da diese in vielen Bereichen von *CleverMail* verwendet werden. Hierbei ist besonders auf die Konsistenz zu achten, da Änderungen an dieser Spezifikation weitreichende Folgen haben können.

Eine weitere Herausforderung stellt die Integration in die verschiedenen Anwendungen im *clevercure* Ökosystem dar, da diese einerseits alle in Java implementiert wurden jedoch auf unterschiedlichen Plattformen laufen und sich daher auch die unterstützten Technologien und Frameworks stark unterschieden. Das Konzept von *CleverMail* stellt eine gute Basis dar auf die man aufbauen kann. Ein Prototyp sollte hier implementiert werden damit die vorgestellten Konzepte auf ihre Tauglichkeit getestet werden können. Abschließend sei angemerkt das anfangs nicht angenommen wurde dass sich so viele Teile von *CCMail* in *CleverMail* wiederfinden werden. Dies wird aber das Wechseln von *CCMail* auf *CleverMail* erleichtern und nur als positiv anzusehen.

Abstract

During the analysis of the existing application *CCMail*, many mistakes has been discovered, which lead to the fact that this application is not capable of meeting the new requirements. The software design and the implemented database schema weren't maintained for tool long and didn't adapt to the new standards, which lead to the fact that an refactoring is now equal to an re-implementation. Nevertheless some features of *CCMail* were adapted to *CleverMail*. Especially the email sending process were adapted, except for the new introduced features. The concept of *CleveeMail* introduced new possibilities which are able now by using the JEE7 platform. The management of the template parameters will be a great challenge, because they are used in many aspects of the newly introduced *CleverMail* application. Especially the consistency of these parameters needs to be ensured, because changes made on the template parameter specification could cause far-reaching impact on the application.

Another challenge will be the integration into the other applications part of the *clevercure* ecosystem, because they are implemented in Java but run on different platforms where the supported and available technologies and frameworks will differ. The concept of *CleverMail* represents a good basis which to build on. A prototype should be implemented which shall be used to test the introduced concepts for usability. Last but not least it should be mentioned that it weren't expected to find some much of *CCMail* in *CleverMail*. This will facilitate the move from *CCMail* to *CleverMail* and is considered to be a positive side effect.

Kapitel 1

Einleitung

Die vorliegende Sachlage beschäftigt sich mit der Konzeption einer neuen Mail-Anwendung, welche in weiterer Folge als *CleverMail* bezeichnet wird, die eine bestehende alte Mail-Anwendung, in weiterer Folge *CCMail* genannt, ersetzen soll. Dieses Konzept wird für das Unternehmen *curecomp* erstellt. Einleitend wird das Unternehmen *curecomp* und dessen Anwendungen vorgestellt.

Das Unternehmen *curecomp* ist ein Dienstleister im SRM-Bereich (*Supplier-Relationship-Management*) und betreibt eine Softwarelösung namens *clevercure*, dessen Komponenten aus den folgenden Anwendungen besteht:

- *CleverWeb* ist eine Web-Anwendung für den webbasierten Zugriff auf clevercure.
- *CleverInterface* ist eine Schnittstellen-Anwendung für die Anbindung der ERP-Systeme der Kunden und Lieferanten, deren Daten mittels XML-Dateien import und exportiert werden können.
- *CleverSupport* ist eine Web-Anwendung, die zur Unterstützung der *Support*-Abteilung dient.
- *CleverDocument* ist ein Dokumentenmanagementsystem für die Verwaltung aller anfallenden Dokumente.

Alle diese Anwendungen erfordern den Versand von *E-Mails*, um verschiedene Systemzustände und Benachrichtigungen den BenutzerInnen mitzuteilen wie z.B.:

- Fehlermeldungen,
- Statusänderungen bei Bestellungen (erstellt, geliefert, storniert, ...),
- Lieferverzugsmeldungen,
- Registrierung eines neuen Lieferanten.

Es sind durch die Kunden und das Unternehmen *curecomp* neue Anforderungen an *CCMail* gestellt worden, die sich nicht mehr in *CCMail* umsetzen lassen. Dies ist begründet in dem Design und der Implementierung von *CCMail*. Diese Arbeit befasst sich einerseits mit der Diskussion des Designs und der Implementierung von *CCMail* und liefert andererseits ein Konzept für *CleverMail*.

Vor der Erstellung dieses Konzepts, wird die bestehende Anwendung *CCMail* , insbesondere deren Design und Implementierung diskutiert, damit aufgezeigt werden kann, welche Designentscheidungen und Implementierungsdetails ein Erweitern von *CCMail* verhindern. In dem Konzept für *CleverMail* sollen die in *CCMail* gemachten Design- und Implementierungsfehler berücksichtigt werden, damit *CleverMail* auch zukünftig neuen Anforderungen gewachsen ist und sich diese neue Anforderungen ohne größere Probleme und Durchführungsaufwand 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 sehr gut vorbereiten.

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

- *Java-JDK-8* (Java-Development-Kit in der Version 1.8),
- *JEE-7-Platform* (Java-Enterprise-Edition Plattform in der Version 7),
- *DB2* (Proprietäre relationale Datenbank von IBM),
- *Wildfly* (RedHat-Applikationsserver, früher bekannt als JBoss-Application-Server).

Über die Zeit haben sich die Anforderungen an *CCMail* so drastisch geändert, dass diese nicht mehr in *CCMail* integriert werden können. Wie bereits erwähnt, liegt dies vor allem am Design von *CCMail*. *CCMail* wurde im Jahr 2002 in *Java 1.4* implementiert und hatte daher nicht die technischen Möglichkeiten, die heute zur Verfügung stehen. Zur Erinnerung: *Generics* stehen erst seit der Version *Java 1.5* zur Verfügung. Bis heute wurden Änderungen in *CCMail* vorgenommen, die keine technologischen Weiterentwicklungen von Java berücksichtigten. Aus heutiger Sicht scheint eine Erweiterung von *CCMail* alleine schon wegen dem großen technologischen Unterschied der Java-Versionen 1.4 und 1.8 sinnlos.

Technologische Weiterentwicklungen fanden zwar in in den anderen Anwendungen wie *CleverWeb* und *CleverInterface* statt, jedoch scheint es so, dass *CCMail* hier vernachlässigt wurde, was dazu geführt hat, dass ein großer technologischer Unterschied zwischen den Anwendungen entstanden ist. Da-

her wurde die Entscheidung getroffen, *CCMail* durch *CleverMail* zu ersetzten, wobei folgender Bachelorarbeit die Grundlage dafür erarbeiten soll.

Im Kapitel 2 wird die alte Mail-Anwendung *CCMail* kritisch betrachtet und analysiert. Folgende Aspekte von*CCMail* finden dabei Beachtung:

- Systemaufbau,
- E-Mail-Versand,
- Software-Design,
- und die Persistenz.

Die erarbeiteten Ergebnisse werden in weitere Folge dazu verwendet um das Konzept für die neue Mail-Anwendung *ClverMail* zu erstellen.

Das Konzept für *CleverMail* wird im Kapitel 3 auf Grundlage der Betrachtungen, die im Kapitel 2 erarbeitet wurden, erstellt. Dabei werden auch die neuen Anforderungen, die an die Mail-Anwendung gestellt wurden, berücksichtigt. Das erstellte Konzept wird Möglichkeiten für die Implementierung von *CleverMail* aufzeigen. Dabei werden folgende Aspekte behandelt:

- möglicher Systemaufbau,
- Prozesse,
- und Persistenz.

Das erstellte Konzept für die Umsetzung von *CleverMail* vor allem neue Technologien und *Frameworks* verwenden. Dadurch soll *CleverMail* die heute zur Verfügung stehenden Möglichkeiten bestmöglich anwenden.

Kapitel 2

Die alte Anwendung

In diesem Kapitel wird die alte Anwendung *CCMail* analysiert und diskutiert. Ziel ist es, einen Überblick über diese Anwendung und deren wesentlichsten Aspekte zu liefern, sowie diese Aspekte genauer zu betrachten. Die Ergebnisse dieser Analyse sollen als Grundlage für das neue Konzept dienen, das auch die Integration in die bestehenden Anwendungen berücksichtigen muss. Diese Integration soll mit geringst möglichen Aufwand erfolgen können, da Probleme bei der Integration negative Auswirkungen auf den produktiven Betrieb haben könnten.

2.1 Systemaufbau

Im folgenden wird der Systemaufbau aus der Sicht der Anwendung *CCMail* und dessen Integration in das System über *MailJobs* diskutiert.

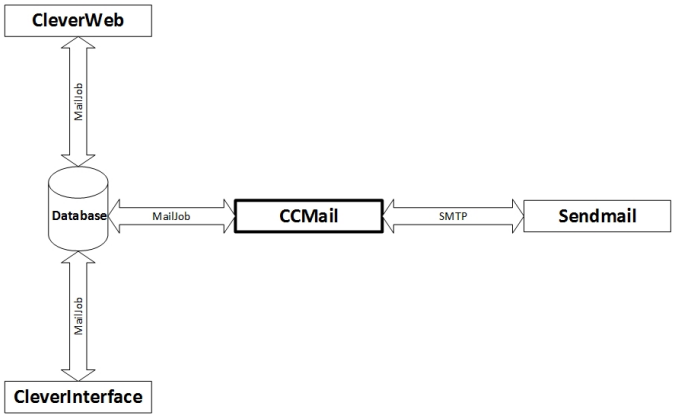

Abbildung 2.1: Systemaufbau und Integration von *CCMail*

Abbildung 2.1 zeigt das Gesamtsystem aus der Sicht der Anwendung *CC-Mail*, wobei anzumerken ist, dass das System bis heute angewachsen ist und nunmehr aus mehreren Anwendungen als wie in Abbildung 2.1 abgebildet besteht. Es lässt sich ableiten, dass das Kernstück des Systems die Datenbank ist. In der Datenbank werden die zu versendenden E-Mail-Nachrichten als sogenannte *MailJobs* verwaltet. Ein *MailJob* ist ein Eintrag in einer Datenbanktabelle namens *MAIL_JOBS*, die alle Informationen einer *E-Mail* enthält. Es sind dabei die eigens implementierten Datenbankzugriffsschichten der einzelnen Anwendungen zu kritisieren, die zwar den Datenbankzugriff kapseln, jedoch nur für jede Anwendung an sich und nicht über Anwendungsgrenzen hinweg, was durchaus möglich wäre. Scott W.Ambler und Parmod J.Sadalge schreiben in ihrem Buch [4, S. 27] treffend:

> *The greater the coupeling, the harder is to refactore something. This is true of code refactoring, and it is certainly true of database refactoring*

Da jede Anwendung ihre eigene Datenzugriffsschicht implementiert, muss jede Anwendung bei einer Datenbankänderung ihre Implementierung anpassen. Eine zentrale Datenbankzugriffsschicht würde nur eine Änderung an einer Stelle erfordern. Also haben wir hier eine Form der starken Koppelung die sich durch die Code-Duplikate ausprägt.

Die Anwendungen *CleverWeb* und *CleverInterface* erstellen über ihre eigens implementierten Datenbankzugriffsschichten *MailJob-Entitäten* in der Datenbank, welche zeitgesteuert von *CCMail* ausgelesen, verarbeitet und in Form von *E-Mails* versendet werden. *CCMail* ist als Konsolen-Anwendung implementiert und enthält alle Ressourcen, die es benötigt, um die *MailJob-Entitäten* zu verarbeiten. Auch hier wirken sich die eigens implementierten Datenbankzugriffsschichten aus, da es keine einheitliche Spezifikation für das Erstellen eines *MailJobs* gibt. Validierungen, ob ein zu erstellender *MailJob* gültig ist, werden den einzelnen Anwendungen überlassen und sind nicht an einer zentralen Stelle umgesetzt. Daher muss sich die Implementierungen in *CCMail* darauf verlassen, dass alle Anwendungen die *MailJobs* korrekt anlegen, damit diese von *CCMail* korrekt verarbeitet werden können.

Als Mail-Server wird *Sendmail* verwendet. Es handelt sich hierbei um eine Anwendung, die für Linux Distributionen frei verfügbar ist. *CCMail* versendet die *E-Mails* über SMTP *(Simple-Mail-Transport-Protocol)* an *Sendmail*, das die *E-Mails* seinerseits an die EmpfängerInnen versendet.

2.2 E-Mail-Versand

Der im folgenden beschriebene Prozess des E-Mail-Versands zeigt auf wie hinsichtlich des Systemaufbaus beschrieben in 2.1 der E-Mail-Versand vom Anlegen eines *MailJobs* bis hin zum Versand der eigentlichen *E-Mail* funktioniert.

Als Kernkomponente des Systems wurde die Datenbank identifiziert, welche die *MailJob-Entitäten* hält, die wiederum von *CCMail* aus der Datenbank gelesen und verarbeitet werden. 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*, durch die Zeitdifferenz zwischen dem Anlegen eines *MailJobs* durch die Anwendungen und dem tatsächlichen Versand der E-Mail-Nachricht durch *CCMail*.

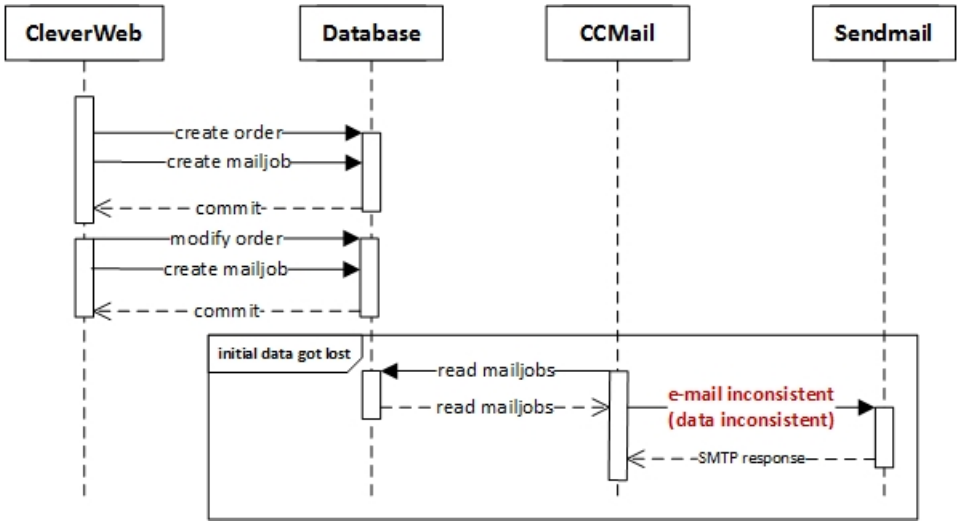

Abbildung 2.2: Gesamtprozess des E-Mail-Versands

Wie man aus dem Sequenz-Diagramm in Abbildung 2.2 ableiten kann, ist eines der Hauptprobleme am E-Mail-Versand die mögliche Inkonsistenz, was in der Art und Weise, wie die *MailJob* Entitäten verarbeitet werden, begründet ist. Aufgrund der zeitgesteuerten bzw. zeitversetzten Verarbeitung kann es vorkommen, dass sich die zugrunde liegenden Daten einer *E-Mail* ändern, bevor diese versendet wurde. In dem Beispiel in Abbildung 2.2 wird eine Bestellung angelegt und kurz darauf geändert. Dies geschieht, bevor die *E-Mail* über das Anlegen der Bestellung versendet wurde. Es wurde zwar ein neuer *MailJob* angelegt, aber beide *MailJob-Einträge* verweisen auf dieselbe Bestellung. Dadurch enthalten beide versendeten *E-Mails* dieselben Daten und die Daten der erstellten Bestellung gingen verloren, da sie durch die gemachten Änderungen überschrieben wurden.

Dies ist begründet in der Art und Weise, wie die *MailJob-Einträge* aufgebaut sind. Ein *MailJob* hält die Daten für den Versand einer E-Mail, wobei hierbei nicht die gesamte *E-Mail* oder die verwendeten Daten gespeichert werden, sondern lediglich die Parameter, die in einer SQL-Abfrage *(Structured-Query-Language)* verwendet werden, um die Daten für die *E-Mail* zu erhalten. Sollten sich also die Datenbank-Entitäten der involvierten Tabellen ändern, so sind die ursprünglichen Daten nicht mehr wiederherstellbar. Dadurch ist auch ein erneuter Versand einer bereits versendeten *E-Mail* nicht mehr möglich bzw. es kann nicht garantiert werden, dass diese *E-Mail* dieselben Daten enthält wie beim ersten Versand.

Ein weiteres Problem liegt in der zeitgesteuerten Verarbeitung der *MailJobs* durch *CCMail*. Lange wurde nicht geprüft, ob bereits ein *CCMail-Prozess* gestartet wurde, bevor dieser erneut gestartet wird. Dies hat dazu geführt, dass es vorkam ,dass mehrere Prozesse gleichzeitig die *MailJob-Entitäten* verarbeiten und daher die *E-Mail* mehrmals versendet wurden. Dieses Problem ist begründet durch die Tatsache, dass in Verarbeitung stehende *MailoJob-Entitäten* nicht als "*In Progress*"markiert wurden und von parallel laufenden Prozessen ausgelesen und verarbeitet wurden. Nun wird zwar geprüft, ob bereits ein Prozess gestartet wurde, bevor ein neuer Prozess gestartet wird, um zu verhindern das parallel laufende Prozesse auftreten. Dies macht es aber unmöglich die Arbeit auf mehrere Prozesse aufzuteilen. Der Ansatz die *E-Mails* in nur einem Prozess zu verarbeiten, hat zur Folge dass der E-Mail-Versand seriell verläuft, obwohl angemerkt sei, dass die einzelnen Nachrichten sehr wohl parallel in eigenen *Threads* innerhalb des Prozesses verarbeitet und versendet werden. Man könnte die Arbeit auf mehrere Prozesse aufteilen und so die Performance verbessern und den Zeitaufwand für den Versand minimieren.

2.3 Software-Design

Nachdem der Systemaufbau diskutiert wurde befassen wir uns jetzt mit dem Software-Design von *CCMail*. *CCMail* wurde als Konsolen-Anwendung implementiert und stellt alle Ressourcen, die zur benötigt werden, zur Verfügung, wie:

1. E-Mail-Vorlagen,
2. Datenbankabfragen,
3. und die implementierten E-Mail-Typen.

Die Klasse *CCBasicEmail* implementiert die gesamte Funktionalität für den Versand einer *E-Mail* und ist die Basisklasse alle implementierten E-Mail-Typen. Die Klasse *CCMailingDao* implementiert alle Datenbankabfragen über alle E-Mail-Typen hinweg. Diese beiden Klassen enthalten die gesamte

Logik für die Verarbeitung eines *MailJob* und des Versand einer *E-Mail*.

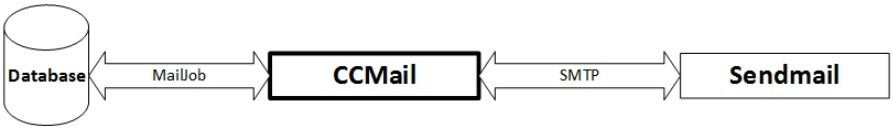

Abbildung 2.3: Teilsystem *CCMail*

Der folgende Abschnitt wird die Schwächen der bestehenden Implementierung und deren Design analysieren. Die Ergebnisse dieser Analyse müssen bei der Erstellung des neuen Konzeptes mit einfließen und verhindern dass bereits gemachte Fehlentscheidungen sich wiederholen, sowie gute Ansätze weiterverfolgt werden.

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

1. *CCBasicEmail* ist die Basisklasse aller E-Mail-Typen, die als abgeleitete Klassen von *CCBasicEmail* implementiert wurden. Sie enthält alle bereitgestellten Funktionalitäten.
2. *CCMailingDao* ist die Schnittstelle zur Datenbank, welche alle SQL-Abfragen über alle E-Mail-Typen hinweg enthält
3. *CCMailingFactory* ist die *Factory-Method-Klasse* für das Erstellen von *CCMailingDao* Objekten.

2.3.1 Klasse *CCBasicEmail*

Einleitend wird die Vererbungshierarchie der Klasse *CCBasicEmail* diskutiert, welche die Basisklasse aller E-Mail-Typen darstellt.

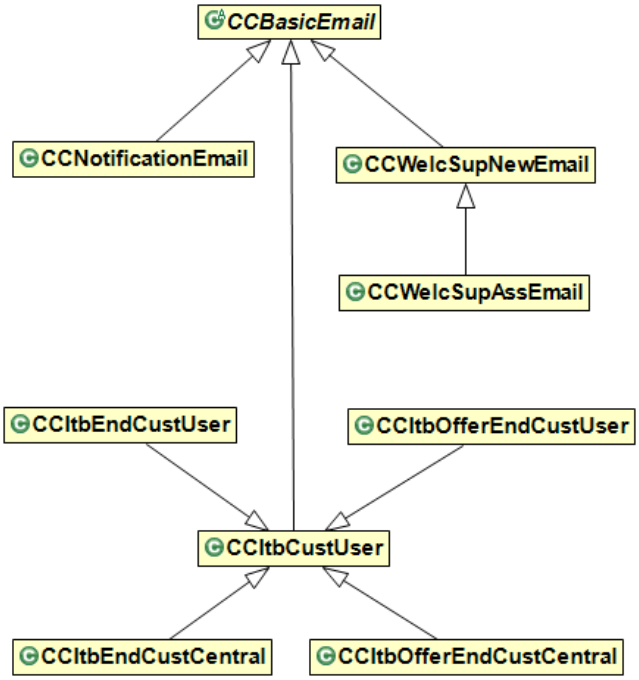

Abbildung 2.4: Auszug aus der Vererbungshierarchie von *CCBasicEmail*

Aus dem Klassendiagramm in Abbildung 2.4 lässt sich ableiten, dass die einzelnen E-Mail-Typen als eigene Klassen abgebildet wurden. Somit ist jeder E-Mail-Typ auch als eigene Java-Klasse abgebildet.

Am Beispiel der Klasse *CCItbCustUser* ist ersichtlich, dass neben dem Abbilden eines E-Mail-Typs als eigene Java-Klasse man ebenfalls eine eigene Subvererbungshierarchie eingeführt hat, um E-Mail-Typen, die sich in einem gemeinsamen Kontext befinden, zu gruppieren. Ob dies ein guter Ansatz ist, um kontextabhängige Ressourcen zu gruppieren, ist zu hinterfragen. Es gäbe hier andere Ansätze, wie man eine solche Gruppierung hätte realisieren können, die flexibler sind als eine Vererbungshierarchie. Eine Vererbungshierarchie ist starr und Änderungen an der Struktur können sich negativ auf die Gesamtstruktur auswirken. Ebenso produziert man so eine Vielzahl von Klassen, die gewartet werden müssen und die Struktur einer Subvererbungshierarchie lässt sich nur über ein Klassendiagramm darstellen und ist nicht aus dem Quelltext abzuleiten. Ebenso wird man bei den Vererbungshierarchien schnell an Grenzen stoßen, da hier nur gerichtete Graphen

möglich sind und Mehrfachvererbung bei Klassen von Java nicht unterstützt wird.

Mehrfachvererbung, auch wenn unterstützt, ist aber ohnedies zu vermeiden, da hier Kollisionen bei den Klassenvariablen und Methoden auftreten können. Außerdem wird durch Mehrfachvererbung die Komplexität der Klassenhierarchie nur unnötig erhöht und bringt daher keine Erleichterungen mit sich.

2.3.2 Klasse CCItbCustUser

Nachdem die Vererbungshierarchie von *CCBasicEmail* diskutiert wurde, wird im Folgenden als Beispiel einer Implementierung von *CCBasicEMail* die Implementierung der Klasse *CCItbCustUser* angeführt. Diese Implementierung dient als Beispiel für die restlichen E-Mail-Typ-Implementierungen, die nach dem selben Prinzip mit ähnlichem Umfang implementiert wurden. Im Abschnitt 2.3.1 wurde behauptet, dass diese Ableitungen eingeführt wurden, um E-Mail-Typen zu gruppieren. Man könnte aber auch annehmen, dass diese eigene Subvererbungshierarchie eingeführt wurde, um gemeinsame Funktionalitäten für die abgeleiteten E-Mail-Typen zu kapseln.

Folgender Quelltext illustriert, dass die Implementierungen der einzelnen E-Mail-Typen hauptsächlich aus dem Erstellen der *E-Mails* besteht, da der Versand bereits in der Klasse *CCBasicEmail* implementiert wurde. Die Parameter für die Vorlage werden aus dem Resultat der spezifischen SQL-Abfrage in der Methode *getMailBody* extrahiert und in der Nachricht bzw. der verwendeten Vorlage verwendet. Die erstellte Nachricht wird dann als Resultat geliefert. Das unterschiedliche Erstellen der *E-Mails* ist also der Grund für das Abbilden der einzelnen E-Mail-Typen als eigene Java-Klassen. Dieser Ansatz produziert viele Klassen, die in einer starren Hierarchie gebunden sind. Und dies nur um das Erstellen der eigentlichen E-Mail-Nachricht in einer eigenen Java-Klasse zu kapseln. Es sei angemerkt, dass diese Klassen auch dazu verwendet um die E-Mail-Typen zu aktivieren oder zu deaktivieren. Zu kritisieren ist hierbei, dass das Erstellen einer *E-Mail* zu stark an einen E-Mail-Typ gekoppelt ist und es hier an Abstraktion fehlt. Die *E-Mail* werden immer nach dem selben Schema erstellt. Es gibt lediglich folgende Unterschiede:

- SQL-Abfrage, welche die Daten aus der Datenbank bezieht.
- Die zugrunde liegende E-Mail-Vorlage.
- Die Paramter für die zugrunde liegende E-Mail-Vorlage.
- Der eindeutige Schlüssel, der den E-Mail-Typ identifiziert.

```
1 public class CCItbCustUser extends CCBasicEmail {
2
3   private Map cache = new HashMap();
4
5   // empty constructor
6   public CCItbCustUser() {
7     super();
8   } // end constructor
9
10  // sets the used dao implementation
11  public CCItbCustUser(CCMailingDAO dao) {
12    super(dao);
13  } // end constructor
14
15  // The unique key for this email type
16  @Override
17  String getMailType() {
18    return "ISCU";
19  } // end getMailType
20
21  // Thread.run method which creates and sends the email
22  @Override
23  public void run() {
24    try {
25      sendEmailNoAttachement(getDAO().getItbStartCustUserMailText());
26    } catch (DAOSysException ex) {
27      LOG.error("DAOSysException in CCItbCustUser.run: ", ex);
28    } finally {
29      stopMe();
30    } // end try-catch-block
31  } // end run
32
33  // Method which creates the email body
34  @Override
35  protected String getMailBody(String bodyKey, String bodySQLKey)
36    throws DAOSysException {
37    int lanId   = ((CCItbVO)currVO).getLanguageId();
38    int itbhId  = ((CCItbVO)currVO).getItbhID();
39    String body = "";
40    String key  = itbhId + "_" + lanId;
41    if (cache.containsKey(key)) {
42      body = (String)cache.get(key);
43      LOG.debug("48: Got from cache key: " + key
44            + " body: " + body);
45    } else {
46      Object[] allParams = getDAO().getItbCustData((CCItbVO)currVO, 19);
47      MessageFormat form = new MessageFormat(rb.getString(bodyKey)
48                                                .trim());
49      body               = form.format(params);
50      cache.put(key, body);
51      LOG.debug("48: DB access for the key: " + key
52          + " got body: " + body);
53    } // end if-else
54    return body;
55  } // end getMailBody
56 }
```

Programm 2.1: Implementierung *CCItbCustUser*

Die folgenden drei Methoden werden von den E-Mail-Typ-Klassen implementiert:

1. *getMailType*: zum Bereitstellen eines eindeutigen Schlüssels, der diesen E-Mail Typ identifiziert.
2. *getMailBody*: zum Erstellen der *E-Mail* aus einer Vorlage, welche mit Parametern befüllt wird
3. *run*: Jeder E-Mail-Typ wird in einem eigenen Thread abgearbeitet. Dabei wird entschiedenen welche Art von E-Mail-Versand genutzt wird. *CCBasiEmail* stellt mehrere Implementierungen zur Verfügung z.B.:
   - ohne Anhänge,
   - mit Anhängen, welche über das lokale Filesystem zur Verfügung gestellt werden, und
   - mit Anhängen, welche über externe Systeme zur Verfügung gestellt werden.

Der Quelltext aus Abbildung 2.1 illustriert, dass die E-Mail-Typen keine nennenswerte Logik haben, sondern lediglich für das Erstellen der E-Mail-Nachricht verantwortlich sind.

2.3.3 Klasse 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.

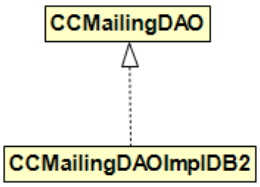

Abbildung 2.5: Vererbungshierarchie von *CCMailingDao*

Die Vererbungshierarchie aus Abbildung 2.5 ist sehr einfach, da man sich hier nicht für eine Aufteilung der Datenbankzugriffsschicht für die einzelnen E-Mail-Typen entschieden hat. Dabei ist zu bemängeln, dass sich alle Datenbankabfragen über alle E-Mail-Typen hinweg befinden und man es versäumt hat hier Schnittstellen einzuführen, welche die kontextabhängigen Datenbankabfragen spezifizieren, also eine Schnittstelle für jeden E-Mail-Typ. Mit einer Aufteilung auf mehrere Schnittstellen hätte man die War-

tung der Datenbankabfragen vereinfacht. Mit dem Ansatz der Aufteilung auf mehrere Schnittstellen, wäre man einerseits gezwungen Präfixe für die Methodennamen einzuführen, da Namenskollisionen sehr wahrscheinlich sind, und andererseits muss man darauf Acht geben, bestehende Implementierungen bei einem Restrukturieren 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 wiederum eigene Ressourcen ein, die sich durch ihren Namen unterscheiden.

Eigene Schnittstellen und Implementierungen je E-Mail-Typ hätten es ermöglicht, für jeden dieser E-Mail-Typen Ressourcen zur Verfügung zu stellen, die nur dieser E-Mail-Typ verwendet. Mann hätte Flexibilität erhalten und hätte sich trotzdem auf eine gemeinsame Basis einigen können.

Der Ansatz, die Implementierungen von *CCMailingDAO* für verschiedene Datenbanken zu zur Verfügung zu stellen, ist an sich gut, jedoch hätte man sich mit der Nutzung von ORM *(Object Relational Mapping)* das Leben erleichtern können, da ein *ORM-Provider*, wie z.B.: *Hibernate*, bereits die zugrunde liegende Datenbank abstrahiert. Datenbank-spezifische SQL-Anweisungen und Funktionalitäten werden zwar von den *ORM-Providern* nicht zur Verfügung gestellt, jedoch sind solche spezifischen Teile in *CCMail* nicht zu finden. Die Entscheidung, sich hier auf native SQL-Abfragen zu stützen, bringt das Problem mit sich, dass die zugrunde liegende Datenbank nicht von der Anwendung abstrahiert ist und man sich so an eine spezielle Datenbankimplementierung bindet.

2.3.4 Klasse *CCMailingDaoFactory*

Zu kritisieren ist auch die Art und Weise wie ein Objekt von *CCMailingDao* erzeugt wird. Man nutzt hier das Softwaremuster *Factory-Method*, jedoch wird statisch die zu verwendende Implementierung in *CCMailingDaoFactory* definiert, was das Austauschen der Implementierung zur Laufzeit unmöglich macht. Man hätte dies konfigurierbar machen sollen, z.B über eine Konfigurationsdatei, die den zu verwendenden Implementierungsnamen zur Verfügung stellt.
Im Buch *Refactoring to patterns* [3, S. 72] wird als Nachteil einer *Factory-Method* die erhöhte Komplexität des Designs genannt, wenn eine direkte Instanziierung auch genügen würde. Nachdem die Instanziierung in der Basisklasse *CCBasicEmail* erfolgt und die Ableitungen die Objekte über eine Get-Methode oder die geschützte Datenkomponente erreichen können, hätte man auf diese *Factory* verzichten können, da die Abstraktion bereits über

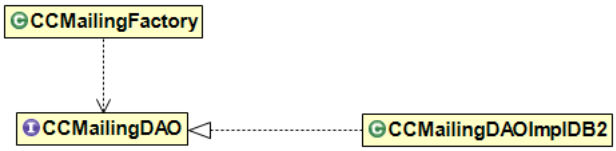

Abbildung 2.6: *CCMailingDaoFactory* für *CCMailingDao*

die Basisklasse *CCBasiEmail* erreicht wurde. Somit ist der Formalparameter der Konstruktoren vom Typ *CCMailingDAO* sinnlos und der Grund warum man dies eingeführt hat, ist nicht ersichtlich. Man hätte die Verwaltung der *CCMailingDAO* Objekte in der abstrakten Basisklasse *CCBaisEmail* halten sollen, ohne die abgeleiteten Klassen damit zu verschmutzen.

Zusätzlich befinden sich die Quelltexte der Schnittstellen zusammen mit ihrer Implementierungen in einem einzigen Projekt. Dies ist auch als ein halbherziger Versuch zu werten, die Implementierung von *CCMailingDao* austauschbar zu machen. Man hätte hier die Quelltexte der Schnittstellen und der Implementierungen auf eigene Projekte aufteilen sollen. Somit hätte man die Abhängigkeit zu den konkreten Implementierungen der Schnittstellen vermieden und hätte sich nicht der Gefahr ausgesetzt, dass ein Entwickler sich direkt auf eine Implementierung beziehen könnte und daher immer gezwungen wäre, mit den Schnittstellen zu arbeiten.

2.4 Datenbank

Abschließend wird hier der Aufbau des Datenbankschemata betrachtet, welches die Kernkomponente des Systems aus der Sicht von *CCMail* darstellt. Es wurde bei diesem Schemata offensichtlich auf Fremdschlüssel verzichtet, was grundsätzlich nur in Spezialfällen anzuraten ist. In dem Buch *Refactoring Database* [4, S. 213] wird als Argument für nicht verwendete Fremdschlüssel die Performance genannt, wobei in diesem Fall diese Begründung nicht standhält. Die beiden Anwendungen *CleverWeb* und *CleverInterface* erstellen lediglich einzelne oder wenige *MailJob* Einträge auf einmal und *CCMail* ist die einzige Anwendung, die diese *MailJob* Einträge einmalig ausliest und verarbeitet. Also sollte in diesem Fall die Performance ohnehin kein Problem darstellen, da hier eine Vielzahl von Teilnehmer und die dadurch resultierende Konkurrenz nicht geben ist. Das Problem von nicht verwendeten Fremdschlüsseln wird in *Refactoring Databases* [4, S. 213] wie folgt beschrieben.

> *The fundemental tradeoff is performance versus quality: Foreign key constraints ensure the validity of the data at the database level at the cost of the constraint being enforced each time the*

> *source data is updated. When you apply Drop Foreign Key, your applications will be at risk of introducing invalid data if they do not validate the data before writing to the database.*

Hier müssen also die Anwendungen selbst die Konsistenz der Daten gewährleisten ansonsten könnten inkonsistente Datenbestände in der Datenbank entstehen, die nachträglich schwer zu identifizieren und zu bereinigen sind. Die Frage ist hierbei ob dieser Ansatz ein guter Ansatz ist?

Wie auch ersichtlich ist wurden die Spalten einer Tabelle mit einem Präfix versehen, der eindeutig über das gesamte Datenbankschema ist. Mann sollte annehmen dass es ausreicht dass die Spaltennamen eindeutig innerhalb des Kontextes einer Tabelle sind und nicht global über das gesamte Datenbankschema. Die Tabellen die neu eingeführt wurden und mit Fremdschlüsseln miteinander verknüpft sind werden dazu verwendet um Datei Anhänge von E-Mail-Nachrichten zu verwalten, die bereits bei der Erstellung des *MailJob* erstellt wurden. Dies war ein Workaround und sollte so auch nicht mehr angewandt werden, da hier die Dateien in Base64 Kodierung verwaltet werden und die Datenbank unnötig mit Daten belasten. Sie sollten in einem File-Storage verwaltet und lediglich referenziert werden, was aber zum Zeitpunkt der Implementierung noch nicht zur Verfügung stand.

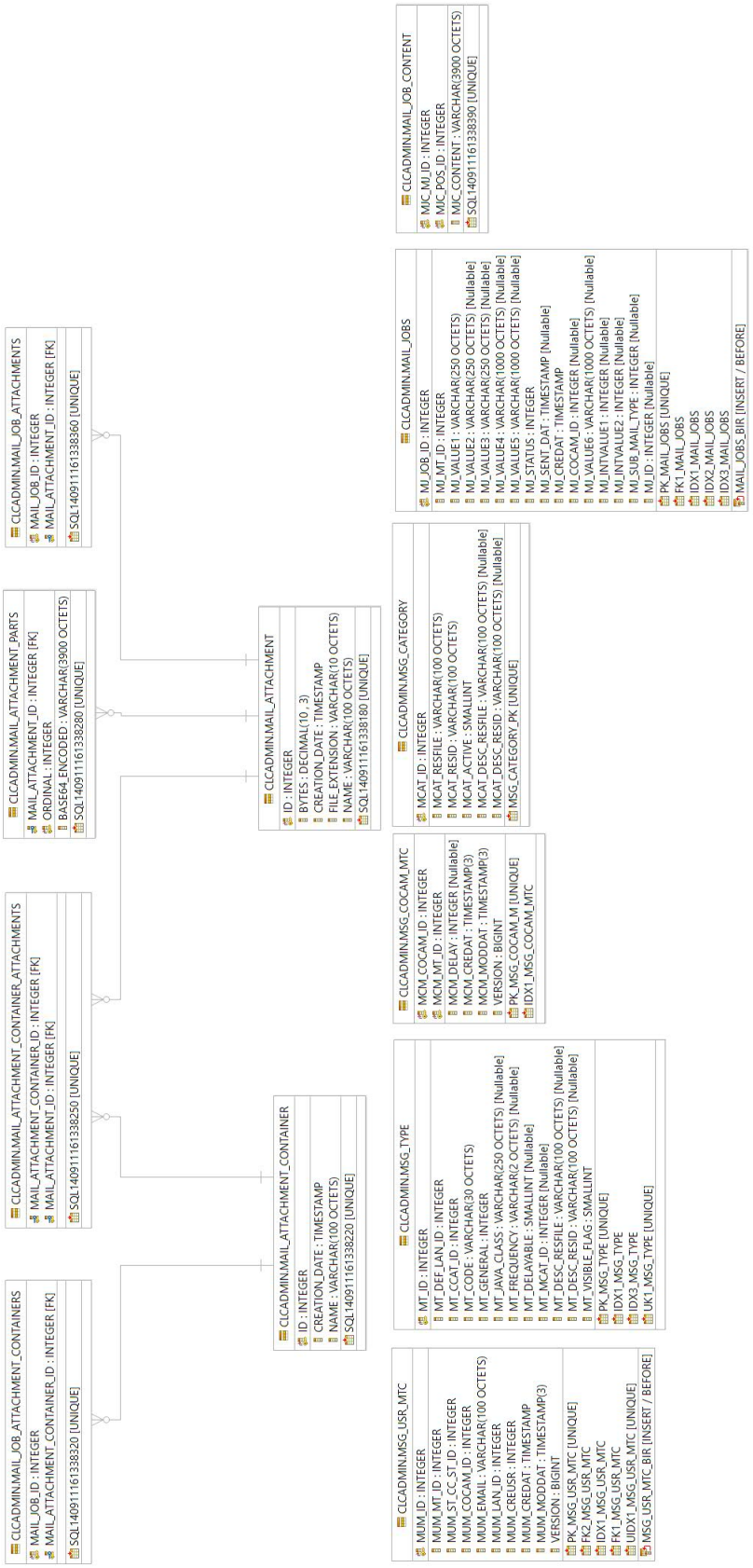

Abbildung 2.7: Datenbankschemata *CCMail*

Kapitel 3

CleverMail

In diesem Kapitel wird nun das Konzept von *CleverMail* behandelt, welches die bestehende Anwendung *CCMail* ablösen soll. Nachdem die Betrachtungen und Analysen von *CCMail* abgeschlossen sind und einige Fehlentscheidungen ausgemacht wurden kann man sich jetzt dem Konzept für *CleverMail* zuwenden. Im Gegensatz zu *CCMail* wird aus der Sicht von *CleverMail* das Gesamtsystem aus mehreren Anwendungen bestehen, die in der Lage sein müssen E-Mail-Nachrichten zu versenden. Über die Zeit ist das Gesamtsystem *clevercure* angewachsen und es wurden neue Anwendungen hinzugefügt, die Aufgrund der Architektur von *CCMyail* nicht eingebunden werden konnten bzw. man sich dazu entschieden hat die Einbindung zu unterlassen.

Folgende Auflistung zeigt alle Anwendungen, die *CleverMail* nutzen werden:

1. *CleverWeb*
   Die Web-Anwendung für den webbasierten Zugriff auf das System
2. *CleverInterface*
   Die Schnittstellen Anwendung für den Datenimport/-export
3. *CleverSupport (neu)*
   Die Web-Anwendung für die Support Abteilung
4. *CleverDocument (neu)*
   Das Dokumentenmanagementsystem welches von allen Anwendungen genutzt wird

Im Gegensatz zu *CCMail* soll *CleverMail* nicht als Konsolenanwendung sondern soll als Java-Enterprise-Anwendung implementiert werden, welche als eigenständige Anwendung in einem Applikationsserver der die JEE7 Plattform Spezifikation unterstützt betrieben werden soll.

Mit der Nutzung der JEE7-Plattform stehen *CleverMail* eine Vielzahl von Features und Frameworks zur Verfügung wie z.B.:

1. JAX-RS 2.0[1]
2. EJB 3.1[2]
3. JPA 2.1[3]
4. JTA 1.2[4]
5. JSF 2.2[5]
6. CDI 1.2[6]
7. uvm.

Diese Features werden es erlauben die Anwendung *CleverMail* so flexibel wie möglich zu gestalten, bringen aber auch ein erhöhtes Maß an Komplexität beim Design mit sich. Martin Fowler führt in seinem Buch *Patterns of Enterprise Application Architecture*[1, S. 5-6] einige Beispiel für Enterprise-Anwendungen an um zu illustrieren dass jede dieser Anwendungen seine eigenen Probleme und Komplexität mit sich bringt und sich daher die Architektur einer Enterprise-Anwendung nicht einordnen und quantifizieren lässt. Daher ist beim Erstellen einer Architektur einer Enterprise-Anwendung die konkrete Nutzung zu berücksichtigen. Der Prozess der Konzeption einer Architektur ist ein kreativer Prozess wobei Konzepte, Best-Practise usw. nur als Unterstützung anzusehen sind und es keinen echten Leitfaden gibt, an den man sich orientieren kann. Die Architektur wird stark von der konkreten Anwendung beeinflusst. Daher kann sich die Architektur je nach Anwendung stark unterschieden.

[1] Java Api for RESTful Web Services 2.0

[2] Enterprise Java Bean. Standard Komponenten für die Entwicklung in Java Enterprise Containern

[3] Java Persistence Api. Java Schnittstelle für Datenbankzugriffe

[4] Java Transaction Api. Java Schnittstelle für den Support von verteilten Transaktionen

[5] Java Server Faces. Java Spezifikation für die Entwicklung von Webanwendungen

[6] Context and Dependency Injection. Java Spezifikation eines IOC-Containers (Inversion of control container)

3.1 Systemaufbau

Im Gegensatz zum Systemaufbau aus der Sicht von *CCMail*, beschrieben in 2.1, soll die Datenbank nicht mehr als Schnittstelle zwischen den Anwendungen und *CleverMail* fungieren. Die Datenbank soll weiterhin ein zentraler Bestandteil von *CleverMail* sein, jedoch soll diese von den Anwendungen abstrahiert werden. Damit erreicht man dass die Anwendungen eine einheitliche Schnittstelle nutzen und nicht ihrerseits eigene Implementierungen warten müssen.

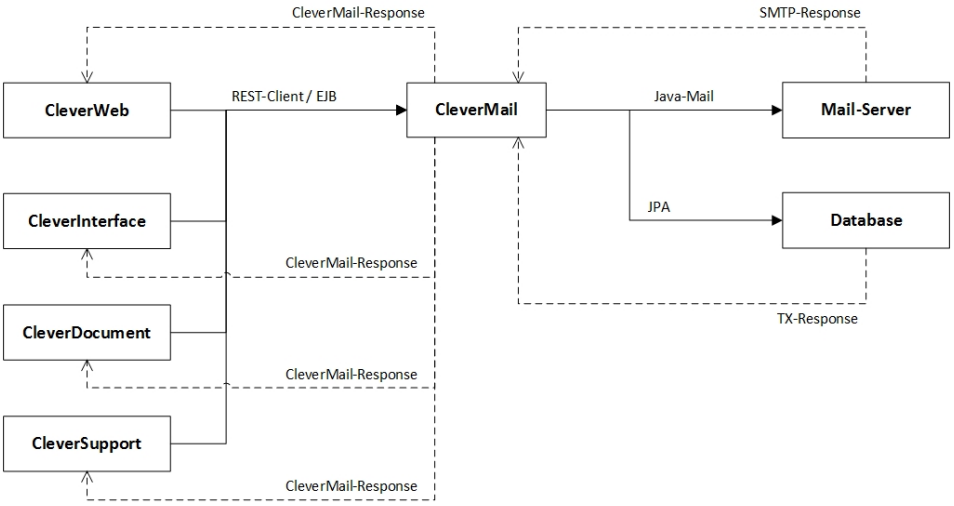

Abbildung 3.1: Systemaufbau und Integration von *CleverMail*

Wie in der Abbildung 3.1 illustriert soll als zentrale Schnittstelle *CleverMail* bzw. dessen implementierte Client-API fungieren wobei diese CLient-API sich wie folgt ausprägen könnte:

1. *REST-Client*
   Eine REST-Schnittstelle zu einen REST-Webservice über den die zur verfügung gestellten Funktionalitäten genutzt werden können.
2. *EJB*
   Ein EJB-Bean welches die zur Verfügung gestellten Funktionalitäten bereitstellt.

CleverMail seinerseits ist für die Persistenzschicht und das versenden der E-Mail-Nachrichten verantwortlich und trennt diese Aufgaben vollständig von den Anwendungen. So kann die Wartung nur an einer Stelle erfolgen und muss nicht über alle Anwendungen hinweg erfolgen. In den Anwendungen würden nur noch Änderungen an den Schnittstellen Eingriffe erfordern.

Dieser Ansatz würde das Problem der eigens implementierten Datenbankzugriffe 2.1 lösen. Ein Problem könnten hier etwaige technologische Unterschiede darstellen wie z.B.:

1. REST nicht verfügbar
2. EJB nicht verfügbar
3. Falscher Source-Level

Obwohl diese Probleme auftreten könnten kann zumindest gewährleistet werden, dass alle Anwendungen dieselbe Schnittstelle und dasselbe Domain-Model verwenden, selbst wenn eigene Implementierungen erforderlich sind. Diese Implementierungen würden eine Softwarekomponente von *CleverMail* darstellen und währen auch Teil dieser Anwendung und dürfen nicht von den Anwendungen selbst bereitgestellt werden. Diese technologischen Unterschiede könnten wie folgt gelöst werden.

1. *REST*
   Integration von JAX-RS 2.0
2. *EJB*
   Integration eines EJB-Containers, zur Verfügung stellen eines Wrappers oder eine eigene Implementierung des spezifizierten Interfaces

3.1.1 REST-Client

Eine REST-Client API, welche sich mit JAX-RS 2.0 einfach realisieren lässt, würde ein hohes Maß an Abstraktion bieten, nur eine geringe Kopplung aufweisen und wenig Abhängigkeiten in der Anwendung erfordern. Dem steht aber gegenüber, dass REST-Services zustandslos sind und sich daher implizit nicht in Datebank Transaktionen einbinden lassen. Dies könnte aber erforderlich sein wenn eine E-Mail nur dann angelegt und versendet werden darf wenn die Transaktion erfolgreich abgeschlossen wurde (z.B.: Anlegen einer Bestellung). Für einen REST-Service startet der Lebenszyklus mit dessen Aufruf und endet mit dem Übermitteln der Response oder wenn die Aktion abgeschlossen wurde (asynchron).

Für diese Problem gibt es eine Lösung in Form eines Konzeptes mit der Bezeichnung Try-Confirm-Cancel (TCC).

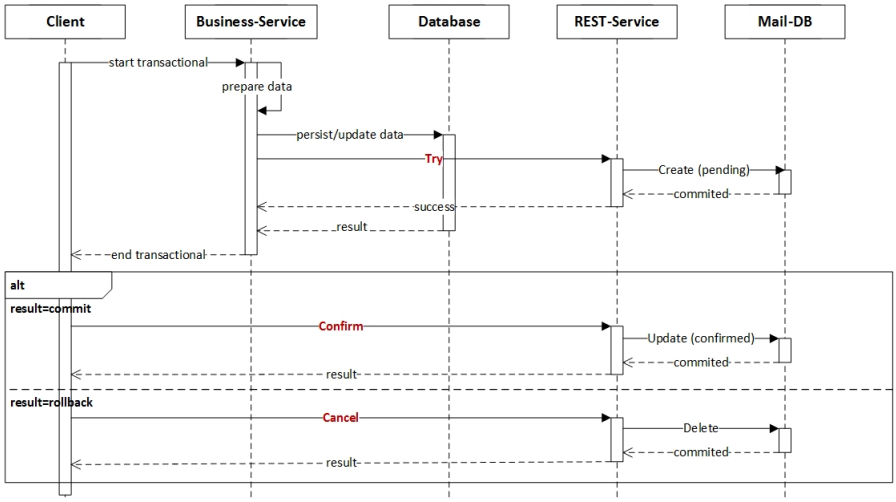

Abbildung 3.2: Beispiel einer Transaktion mit TCC

Mit diesem Konzept müsste der REST-Service den Zustand persistent halten aber als *unconfirmed* markieren, damit dieser in keiner Verarbeitungslogik miteinbezogen wird. Nachdem erfolgreichen Abschluss der Transaktion auf der Client-Seite muss der Client den durch den REST-Service persistent gehaltenen Zustand bestätigen und im Falle eines Fehlers abbrechen. Dies würde zwei Aufrufe zu REST-Services verursachen. Ebenfalls sollte die Transaktion über einen Transaktionskoordinator auf der REST-Seite kontrolliert werden, was wiederum einen Mehraufwand bedeutet. Dieser Transaktionskoordinator ist dafür verantwortlich die REST-Services, die Teil einer logischen Transaktionen sind, zu managen. Mit *TCC* besteht auch die Gefahr das eine Heuristic-Exception auftritt. Beim Auftreten einer solchen Exception würden inkonsistente Datenbestände auf der Datenbank entstehen, da es sein könnte dass nicht alle REST-Services ein Rollback durchgeführt haben.

Diese Probleme bedeuten aber nicht dass REST-Services nicht in Frage kommen. Lediglich für transaktionale Operationen scheinen sie ungeeignet bzw. der Aufwand der betrieben werden muss zu hoch. Ein weiteres Problem könnte die Erreichbarkeit dieser REST-Services sein. Sollte dieser einmal ausfallen oder im Falle eines Deployments nicht erreichbar sein so müsste man eine Rückversicherung haben und die zu erstellenden E-Nachrichten anderweitig zwischenspeichern wie z.B.: in Form einer Textdatei, welche die Daten als JSON enthält.

Diese Onlinepublikation [2] beschreibt den Prozess von *TCC* mit mehre-

ren involvierten REST-Services sehr gut und detailliert.

3.1.2 EJB

Sollte eine Anwendung über EJB mit *CleverMail* kommunizieren so würde hierbei eine starke Kopplung und starke Abhängigkeiten entstehen, da hier mehr Ressourcen benötigt werden. Ebenso könnte im Gegensatz zu einem REST-Client keine eigene Datenbank genutzt werden, da hier ein Zweiphasen-Commit erfolgen müsste. Natürlich würde diese Möglichkeit bestehen aber auch hier wäre man der Gefahr von Heursitc-Exceptions ausgesetzt.

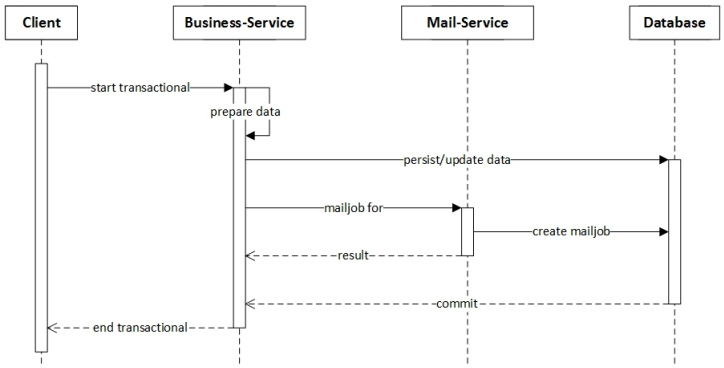

Abbildung 3.3: Beispiel einer EJB (JTA) Datenbanktransaktion

In diesem Fall würde das Anlegen einer E-Mail in derselben Transaktion erfolgen und würde daher auch im Falle eines Roolbacks entfernt werden. Dies ist sicher die angenehmste Art und Weise um E-Mails anzulegen, da hier keine besonderen Mechanismen implementiert werden müssten um die Datenkonsistenz zu gewährleisten. Hierbei währen die E-Mail-Nachrichten auch Teil einer wohl definierten Transaktion.

Wie hier 3 angemerkt können technologische Probleme auftreten wenn z.B.: eine Anwendung kein *EJB* und/oder *JPA* unterstützt. Hierbei müsste man eine eigene Implementierung erstellen, die zumindest Teil von *CleverMail* sein sollte.

Dieser transaktionale Ansatz unterschiedet sich nicht von dem in *CCMail* bereits implementierten jedoch sollen hier die von *CleverMail* zur Verfügung gestellten implementierten Ressourcen verwendet werden. Diese Ressourcen sollten auch im Backend der REST-Services von *CleverMail* angewandt werden, da auch hier die Persistenz der E-Mails gewährleistet werden muss.

3.2 Prozesse

Diese Kapitel behandelt die Prozessspezifikationen von *CleverMail*. Hierbei wird das Augenmerk auf den Mailversand gelegt. Grundlegend wird sich der E-Mail-Versand dadurch unterschieden, dass hier mehrere Ebenen involviert sind, bevor eine E-Mail-Nachricht bereit zum Versand ist. Vom grundlegenden Konzept wird sich gegenüber *CCMail* nicht viel ändern. Es soll immer noch E-Mail-Typen geben, die aber jetzt nicht nur intern konfigurierbar sein sollen sondern ebenfalls durch die Kunden selbst konfiguriert und gesteuert werden können. Hierbei sollen folgende Konfigurationsmöglichkeiten zur Verfügung stehen.

1. Definition von Schedules
2. Definition eigener E-Mail-Vorlagen
3. Konfiguration der Steuerbarkeit von E-Mail-Typen durch Lieferanten (darf aktivieren/de-aktivieren)
4. Definition eines Haftungsausschluss
5. Definition von Standard Datei Anhängen
6. Steuerbarkeit von E-Mail-Typen für spezifischen Lieferanten
7. Konzernübergreifende Konfiguration
8. Definition eigener E-Mail-Typen
9. Konfiguration der Historie der E-Mail-Nachrichten

Zur Zeit stehen diese Features, wenn vorhanden, nur intern zur Verfügung. Die Kunden haben lediglich die Möglichkeit einzelne E-Mail-Typen zu aktivieren oder zu de-aktivieren. Diese E-Mail-Typen können aber mehrere E-Mail-Nachrichten beinhalten. Das Grundlegende Ziel ist dass die Kunden mehr Kontrolle und Konfigurationsmöglichkeiten über die zur Verfügung gestellten E-Mail-Typen erhalten. Es wird hier aber auch solche E-Mail-Typen geben, bei denen diese Konfigurationsmöglichkeiten eingeschränkt werden. Nichts desto trotz soll der Kunde in der Lage sein, den E-Mail-Verkehr seiner E-Mail-Nachrichten besser zu steuern.

3.2.1 E-Mail-Versand

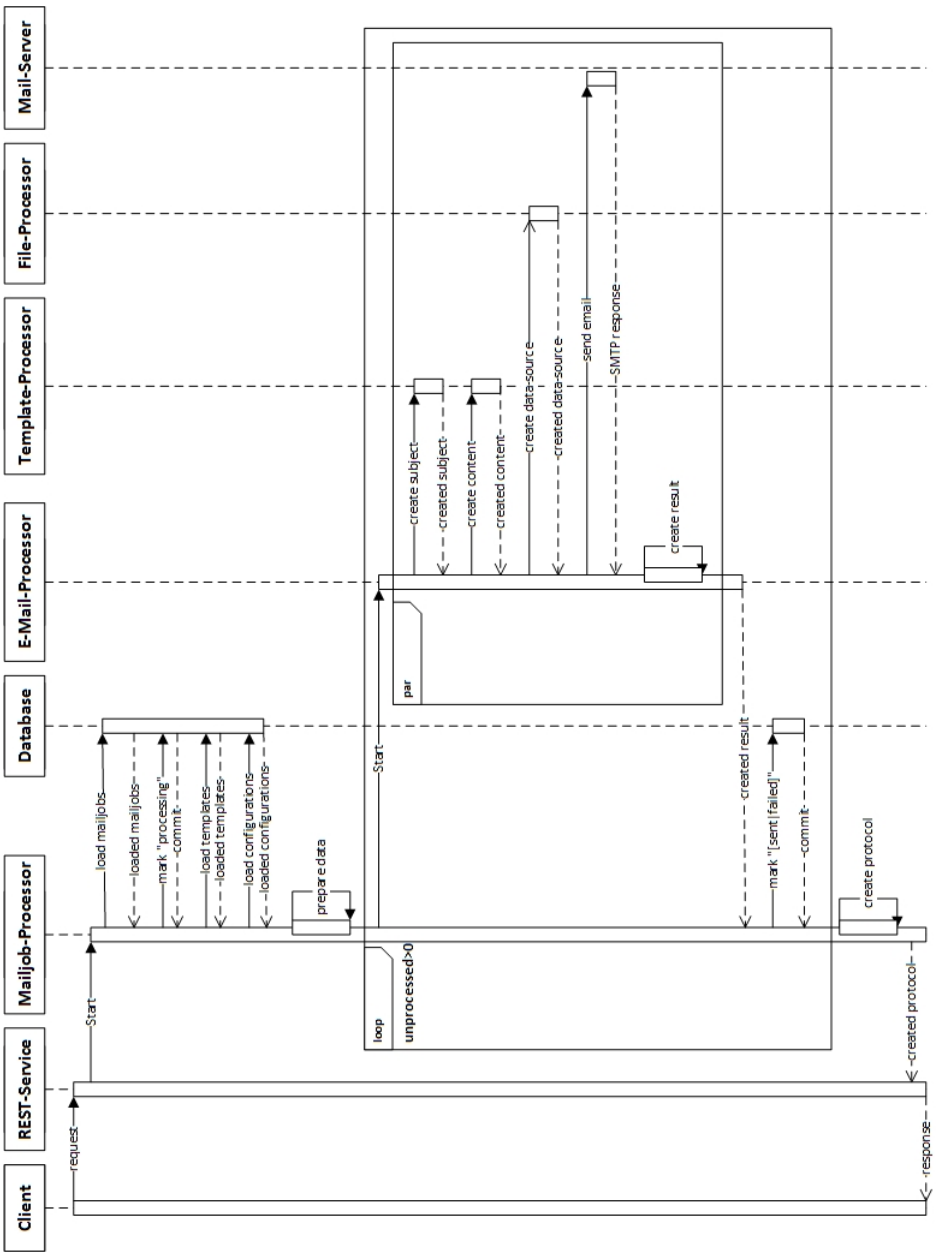

Abbildung 3.4: Prozess des E-Mail-Versands

Die Abbildung 3.4 soll den Prozess des E-Mail-Versands veranschaulichen und illustriert die Softwarekomponenten, die involviert sind um eine E-Mail-Nachricht zu erstellen. In diesem Beispiel wird der Prozess über einen REST-Service gestartet, der in diesem Fall den Prozess synchron abarbeitet und anschließend eine Response in Form eines erstellten Reports zurückliefert. Der REST-Service ist hierbei als optional anzusehen, da dieser Prozess auch anderweitig ausgelöst werden könnte. Nachdem sich dieser Prozess nicht in einer Transaktion eines Client befinden muss würde sich hier eine REST-Schnittstelle anbieten.

Folgende sei der Prozess in grobe Schritte unterteilt angeführt und beschrieben.

Daten Aufbereitung

Nachdem der Prozess gestartet wurde sollen die zu verarbeitenden E-Mail-Nachrichten aus der Datenbank ausgelesen und als *Processing* markiert werden. Im Punkt 2.2 wurde angemerkt, dass das Problem bestand, dass die in Verarbeitung stehenden *MailJob* Entitäten nicht als solche markiert wurden und daher eine parallele Verarbeitung durch mehrere Prozesse nicht möglich war. Dieses Problem besteht in diesem Fall nun nicht mehr. Nachdem die *MailJob* geladen wurden sollen die zugrunde liegenden E-Mail-Vorlagen geladen werden. Hierbei würde sich ein Cache-Mechanismus auszahlen, da die E-Mail-Vorlagen Versionen definieren sollen und sich innerhalb einer Version nicht mehr ändern dürfen. Anschließend sollen die kundenspezifischen Konfigurationen geladen werden. Aus diesen Daten sollen Modelle erstellt werden, die in weitere Folge dazu verwendet werden sollen die E-Mail-Nachrichten zu erstellen.

Erstellen der E-Mail-Nachrichten

Aus den Modellen sollen die E-Mail-Nachrichten erstellt werden. Hierbei wird dieser Prozess in drei Schritte unterteilt.

1. *Erstellen des Betreff*
   Der Betreff wird aus einer E-Mail-Vorlage erstellt und mit Daten befüllt wenn in der Voralge Parameter definiert wurden
2. *Erstellen der Nachricht*
   Die Nachricht wird aus einer E-Mail-Vorlage erstellt und mit Daten befüllt wenn in der Voralge Parameter definiert wurden
3. *Erstellen der DataSoruces*
   Sollten Datei Anhänge definiert worden sein, so werden diese in Form von einer oder mehrerer DataSoruce Instanzen in die E-Mail-Nachricht angefügt.

Die Verarbeitung der E-Mail-Vorlagen soll in einer Komponente mit dem Name *Template-Processor* erfolgen. Nachdem die Werte der verwendeten Vorlagenparameter beim Erstellen eines *MailJob* in der Datenbank gespeichert wurden können diese hier angewandt werden.

Die Verarbeitung der Datei Anhänge soll in einer Komponente mit dem Namen *File-Processor* erfolgen, die die verlinkten Datei Anhänge in Form von *DataSource* Instanzen der E-Mail-Nachricht zur Verfügung stellt. Beim Versand würde die E-Mail-Nachricht den Datei Anhang über diese *DataSource* Instanz laden. Hierbei sollen die Dateien in Form von Links aus dem Dokumentenmanagementsystem *CleverDocument* geladen werden. Es sollen keine Dateien in Form von Base64-Zeichenketten in der Datenbank gespeichert werden, um die Datenbank nicht mit unnötigen Daten zu belasten. Es könnten hierbei verschiedene *DataSource* Implementierungen zur Verfügung gestellt werden, die die Dateien aus verschiedenen Quellen über die verschiedensten Protokolle laden können (z.B.: REST, SOAP, HTTP, usw.).

E-Mail versenden

Der E-Mail-Versand sowie das Erstellen der E-Mail-Nachricht sollte hierbei asynchron erfolgen, da eine sequenzielle Verarbeitung die Performance negativ beeinflussen würde. Nach dem erfolgreichen oder fehlgeschlagenen Versand einer E-Mail-Nachricht soll dieser Status in Form eines Resultates zurückgeliefert werden. Hierbei ist vor allem die SMTP-Response wichtig, die auf jeden Fall gespeichert werden soll, damit der Kunde nachvollziehen kann, wie der E-Mail-Versand seiner E-Mail abgearbeitet wurde und warum die E-Mail nicht zugestellt werden konnte. Ein fehlgeschlagener Versand könnte folgende Ursachen haben:

1. Datei kann nicht geladen werden (nicht vorhanden, Timeout,...)
2. Mail-Server des Empfängers nicht erreichbar
3. E-Mail-Server nimmt Nachricht nicht an
4. uvm.

Es kann unzählige Ursachen haben warum eine E-Mail-Nachricht nicht zugestellt werden kann. Die trifft vor allem auf den E-Mail-Servers des Empfängers zu, auf dessen Konfiguration kein Einfluss genommen werden kann.

3.2.2 E-Mail-Vorlagen(parameter)

Ein weiterer wichtiger Aspekt stellen die Vorlagen und dessen zur Verfügung gestellten Parameter und deren Verwaltung dar. Die Bereitstellung von Vorlagenparametern soll den AnwenderInnen die Möglichkeit bieten auf kontextabhängige Daten in einer E-Mail-Vorlage zugreifen zu können. Dadurch soll die Flexibilität erhöht werden und die AnwenderInnen sollen mehr Freiheiten beim Erstellen einer E-Mail-Vorlage bekommen. Diese Parameter müssen hierbei von den EntwicklerInnen innerhalb eines Kontextes zur Verfügung gestellt werden, wobei hier auch ein Entwicklungsaufwand besteht. Diese Parameter müssen beim Verarbeiten einer E-Mail-Vorlage aus dem aktuellen Kontext ausgelesen werden. Dabei können die Daten aus verschiedenen Objekten oder sogar Objektgraphen kommen. Dies bedeutet dass diese Parameter über eine spezielle Implementierung ausgelesen werden müssen, die auch zukünftig gewartet werden muss. Ebenso werden diese Parameter in verschiedenen Softwarekomponenten verwendet, die sich in verschiedenen Laufzeitumgebungen befinden können. Daher wird es erforderlich sein diese Parameter in einem spezifizierten Model oder über definierte Zuordnungen von einem Model in ein anderes Model zu überführen.

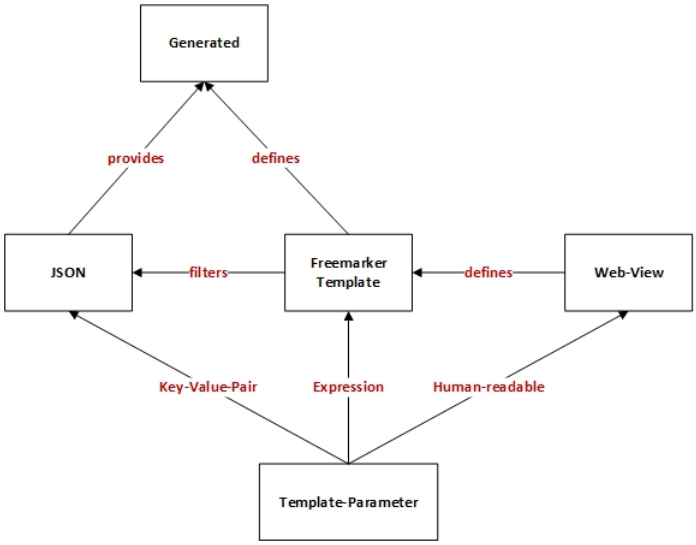

Abbildung 3.5: Verwendung Vorlagenparameter

Wie in der Abbildung 3.5 illustriert werden die Vorlagenparameter in mehreren Kontexten verwendet. Dadurch stellt sich die Frage wie diese Parameter adressiert werden. Man könnte eine Zuordnung in jedem Verwendungskontext erstellen, also jeder Kontext bekommt sein eigenes Model. Dadurch würde man sich zwar lose and die Vorlagenparameter koppeln jedoch erhält man auch eine Model Klasse je Kontext, die gewartet werden muss.

Webseite

Da diese Parameter in E-Mail-Vorlagen verwendet werden wird es auch eine Webseite geben, über die die AnwenderInnen diese Parameter in Ihren E-Mail-Vorlagen verwenden können. Diese müssen natürlich für die AnwenderInnen einen Name bekommen, der natürlich auch in mehreren Sprachen zur Verfügung stehen muss. Dadurch werden diese Parameter auf einen Schlüssel zugeordnet werden müssen, der wiederum auf einen lokalisierten Spracheintrag zugeordnet ist. Die Zuweisung auf diesen Schlüssel ist erforderlich aber benötigt man hier auch nochmals eine Zuordnung auf den Vorlagenparameter?

Freemarker

Ein weiterer Aspekt ist auch die direkte Verwendung der Parameter in der Vorlage selbst. Als Implementierung einer *Template-Enging* soll für die Erstellung der Nachrichten aus den Vorlagen *Freemarker*[7] verwendet werden. Dieses Framework ist ein sehr beliebtes und gut gewartetes Framwork, welches alle benötigten Kontrollstrukturen sowie *Freemarker-Expressions* zur Verfügung stellt. Diese *Freemarker-Expressions* ähneln sehr den Java-EL-Expressions[8]. Die Vorlagenparameter werden in den *Freemarker-Expressions* verwendet, die es ermögliche eine flache Adressierung sowie auch eine Adressierung über Objektgraphen zu definieren. Dadurch stellt sich auch hier die Frage ob diese Expressions wieder über eine Zuordnung zu den Parametern erstellt werden sollen oder ob die Parameter selbst verwendet werden sollen?

JSON

Wenn eine E-Mail in Form eines *MailJob* erstellt wird müssen zum jeweiligen Erstellungszeitpunkt alle Parameterwerte ausgelesen und mit dem *MailJob* persistent gehalten werden. Nachdem die Anzahl der Parameter dynamisch ist und diese Daten lediglich in der Vorlagenverarbeitung verwendet werden ist es hier nicht erforderlich eine eigene Datenstruktur auf der Datenbank zu definieren. Dadurch würde sich hier *JSON* anbieten um diese Daten in die Datenbank zu serialisieren. Nachdem *JSON* eine Objektbeschreibung darstellt und auch über ein *JSON-Schema* spezifiziert werden kann, sollte man überlegen ob man nicht das *JSON* als Spezifikation für die Vorlagenparameter heranzieht. Diese Spezifikation kann in allen involvierten Komponenten verwendet werden.

[7] Frei verfügbare Template-Engine in Java für die Erstellung von dynamischen Textdateien

[8] Java-Expression-Language ist eine Java Spezifikation für Expressions

3.3 Datenbank

Neben der Persistenz der E-Mail-Nachrichten soll ebenfalls die Möglichkeit bestehen, den E-Mail-Versand zu konfigurieren. Dies soll einerseits Firmen intern möglich sein und andererseits auch für die AnwenderInnen. Hierbei sollen folgende Möglichkeiten zur Verfügung stehen:

1. Zeitsteuerung des Versandes
2. Berechtigungen für Modifikationen der E-Mail-Nachrichten
3. Eigene E-Mail-Typen
4. Eigene E-Mail-Vorlagen
5. Zusatzdokumente
6. Speicherverwaltung der E-Mail

Um diese Konfigurationen zu verwalten wird ein Speichermedium benötigt, welches ebenfalls in der Lage ist die Relationen abzubilden. Da bietet sich verständlicherweise eine Datenbank an. Dies wurde bereits in der alten Anwendung *CCMail* so angewandt und soll auch hier fortgeführt werden. Hierbei soll aber das bestehende Datenbankschema nicht berücksichtigt werden und ein neues konzipiert werden. Im Gegensatz zu dem bestehenden Datenbankschema beschrieben im Punkt 2.4 soll hierbei keineswegs auf die referenzielle Integrität zwischen den involvierten Tabellen verzichtet werden. Dies stellt die größte Fehlentscheidung beim Design des Datenbankschemas dar. Des weiteren soll so gut wie möglich auf die Unabhängigkeit des Datenbankschemata geachtet werden. Hierbei ist gemeint dass die Tabellen des Schemata für *CleverMail* nicht in bestehende Tabellen der Anwendung *Clevercure* referenzieren sollen. Damit soll sichergestellt werden, dass dieses Datenbankschema auch als Standalone verwendet werden kann. Sollten Referenzen auf Tabellen wie Benutzer (USER), Kunde (COMPANY) des *Clevercure* Datenbankschema eingeführt werden, so kann *CleverMail* nicht außerhalb des Kontextes von *Clevercure* existieren und würde immer nur für diese Anwendung zur Verfügung stehen. Das Modul *CleverMail* sollte aber auch als Standalone-Anwendung funktionsfähig sein, auch wenn es für die Anwendung *Clevercure* entwickelt werden soll. Diese begründet sich durch die Anwendungsfälle selbst, die nicht auf eine bestimmte Repräsentation von Inhabern von persistenten E-Mail-Nachrichten, E-Mail-Typen oder Konfigurationen angewiesen sind. Mann sollte sich hier die Möglichkeiten für eine zukünftige Anwendung in anderen Anwendungen offen halten anstatt sich den Aufwand einer umfangreichen Refaktorisierung beim Eintreffen eines solchen Falles auszusetzen.

Auch wenn dieser Ansatz verfolgt werden soll ist es natürlich auch möglich diese Referenzen herzustellen solange der Datenzugriff gekapselt nur einer Stelle oder voll definierten Stellen stattfindet. Somit ist gewährleistet,

dass eine Refaktorisierung keine unerwünschten Nebeneffekte hat und man nicht vergisst Änderungen an Stellen nachzuziehen, die vergessen wurden. Dies kann leicht passieren wenn solche Implementierungsteile beim Refaktorisieren nicht bekannt sind. Scott W.Ambler und Parmond J.Sadalage beschreiben dies in Ihrem Buch *Refactoring Databases* [4, S. 66] wie folgt.

> *You could implement the SQL logic in a consistent manner, such as having save(), delete(), retrieve(), and find() operations for each business class. Or you could implement data access objects(DAOs), classes that impleemnt the data access logic separately from business classes.*

Natürlich ist der Ansatz mit *DAOs* vorzuziehen, da hier der Datenzugriff gekapselt an einer Stelle implementiert wird und nicht zerstreut über n-Business-Klassen implementiert wird.

3.3.1 Datenbankschemata CleverMail

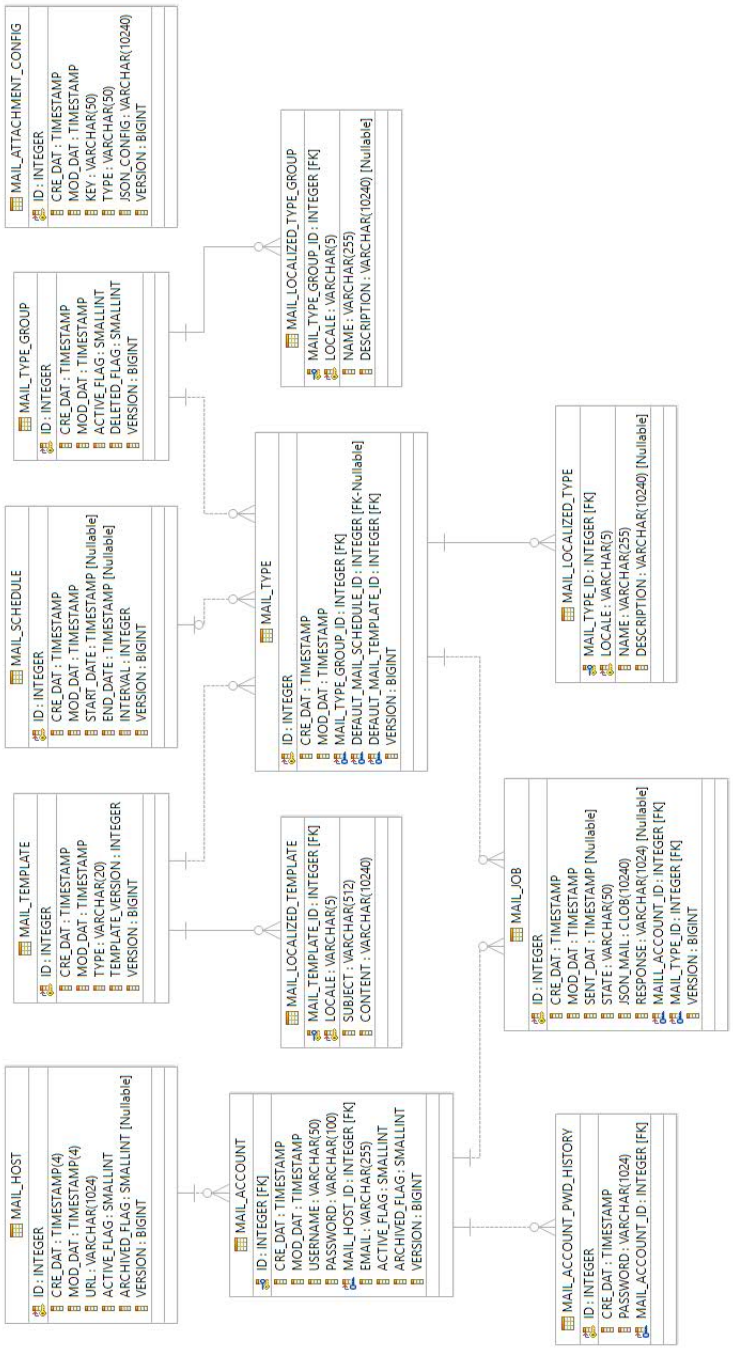

Abbildung 3.6: Datenbankschema CleverMail

Das angeführte Datenbankschema 3.6 stellt eine mögliche Implementierung des Datenbankschemata dar. Dieses Datenbankschema enthält keine Referenzen auf das Datenbankschema von *Clevercure*. Die Integration in das Datenbankschema von *Clevercure* bzw. die Integration des Datenbankschemata von *Clevercure* in das Datenbankschemata von *CleverMail*, wie im diesen Beispiel angedacht, könnte über N-M-Tabellen abgebildet werden. Mit dieser Art der Tabellenrelation kann auch eine 1-N Relation abgebildet werden, wobei noch zusätzlich ein Unique-Constraint auf den Forgein-Key der referenzierten Tabelle hinzugeüfgt werden muss. Mann erhält mit diesem Ansatz zwar noch zusätzliche Relationstabellen, die ebenfalls einen zusätzlichen Join erfordern, jedoch ist so sichergestellt, dass das Datenbankschemata von *CleverMail* unabhängig bleibt.

Es ist auch anzumerken, dass das Datenbankschemata 3.6 von *CleverMail* sehr dem Datenbankschemata 2.7 ähnelt. Wobei natürlich neue Tabellen hinzugefügt wurden um die neuen Features wie die Steuerbarkeit des E-Mail-Versandes zu unterstützen. Nicht desto Trotz finden sind viele Eigenschaften von *CCMail* wieder mit einen etwas anderen Ansatz sowie natürlich auch der Umsetzung.