

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

Erklärung	iii
Kurzfassung	v
Abstract	vii
1 Einleitung	1
2 Die alte Anwendung <i>CCMail</i>	4
2.1 Systemaufbau	4
2.2 <i>E-Mail</i> -Versand	6
2.3 Software-Design	7
2.3.1 Klasse <i>CCBasicEmail</i>	9
2.3.2 Klasse <i>CCItbCustUser</i>	10
2.3.3 Klasse <i>CCMailingDao</i>	12
2.3.4 Klasse <i>CCMailingDaoFactory</i>	13
2.4 <i>CCMail</i> -Datenbank	14
2.4.1 Datenbankschemata von <i>CleverMail</i>	16
3 Die neue Anwendung <i>CleverMail</i>	18
3.1 Systemaufbau	19
3.1.1 1. Möglichkeit (<i>REST-Client</i>)	21
3.1.2 2. Möglichkeit (<i>EJB</i>)	22
3.2 <i>E-Mail</i> -Prozesse	23
3.2.1 <i>E-Mail</i> -Versand	25
3.2.2 <i>E-Mail</i> -Vorlagen und -Parameter	28
3.3 <i>CleverMail</i> -Datenbank	30
3.3.1 Datenbankschemata von <i>CleverMail</i>	32
4 Zusammenfassung	34
Quellenverzeichnis	36
Literatur	36
Online-Quellen	36

Kurzfassung

In der Analyse der bestehenden Anwendung *CCMail* wurden viele Fehlentscheidungen und Probleme identifiziert, die dazu geführt haben, dass *CCMail* den neuen Anforderungen nicht mehr gerecht werden kann. Das Softwaredesign sowie das Datenbankschema wurden zu lange nicht vernachlässigt und haben sich zu lange nicht den neuen Standards und Technologien angepasst, was dazu geführt hat, dass eine Umstrukturierung einer Reimplementierung gleichzusetzen wäre. Trotzdem wurden in das Konzept von *CleverMail* viele Eigenschaften von *CCMail* übernommen. Vor allem was den *E-Mail*-Versand selbst betrifft, der bis auf die neuen Möglichkeiten, beinahe gleich verläuft. Im Konzept von *CleverMail* wurden vor allem neue Möglichkeiten, die mit der Verwendung der JEE-7-Plattform zur Verfügung stehen, berücksichtigt. 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 von *CleverMail* in die verschiedenen Anwendungen im Gesamtsystem von *Clevercure* dar, da die Anwendungen einerseits alle in unterschiedlichen *Java*-Versionen implementiert wurden und andererseits in verschiedenen Laufzeitumgebung betrieben werden und sich daher auch die unterstützten Technologien und Bibliotheken stark unterscheiden. Das Konzept von *CleverMail* stellt eine gute Basis dar. Mit einem implementierten Prototypen können die vorgestellten Konzepte auf ihre Tauglichkeit getestet werden. Abschließend sei angemerkt, dass anfangs nicht angenommen wurde, dass sich so viele Aspekte von *CCMail* sich in *CleverMail* wiederfinden. Dies wird aber das Wechseln von *CCMail* auf *CleverMail* erleichtern und ist als positiv anzusehen. Die wesentlichen Unterschiede zwischen *CCMail* und *CleverMail* sind:

- Die Datenbank ist nicht mehr die Schnittstelle zwischen *CCMail* und den anderen Anwendungen wie z.B. *CleverWeb*,
- Die Integration in andere Softwarekomponenten in anderen Laufzeitumgebungen ist jetzt möglich und
- Die *E-Mail*-Vorlagen sind nicht mehr statisch definiert sondern können

dynamisch erstellt und modifiziert werden.

Vor allem die Möglichkeit der Integration von *CleverMail* in andere Softwarekomponenten ist hervorzuheben, da diese in *CCmail* nicht unterstützt wurde, aber in *CleverMail* ein zentraler Bestandteil ist. Dadurch wird sichergestellt, dass in *CleverMail* alle Implementierungen gekapselt werden und von anderen Softwarekomponenten verwendet werden können.

Abstract

During the analysis of the existing application *CCMail*, many mistakes have been discovered, which lead to the fact that *CCMail* is not capable of meeting the new requirements. The software design and the database schema were neglected for a long time and didn't adapt to new standards, which leads to the fact that a refactoring is equal to an re-implementation. Nevertheless some features of *CCMail* were adopted to *CleverMail*. Especially, the e-mail sending process was adopted, except for the newly introduced features. The concept of *CleeveMail* introduced new possibilities which are feasible 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 have a far-reaching impact on the application.

Another challenge will be the integration into the other application components of the *clevercure* system, because they are implemented in different Java versions and run on different runtime environments, where the supported and available technologies and frameworks differ. The concept of *CleverMail* represents a good basis. An implemented prototype could be used to check the introduced concepts for usability. Last but not least it should be mentioned that it wasn't expected to find so much of *CCMail* in *CleverMail*. This will facilitate the move from *CCMail* to *CleverMail* and is considered to be a positive side effect. Some of the main differences between *CCMail* and *CleverMail* are:

- The database is no longer the interface between *CCMail* and the other applications such as *CleverWeb*,
- The integration into other software components is now possible and
- The e-mail templates are no longer static and can be defined and modified dynamically.

Especially the possibility to integrate *CCMail* into other software components needs to be emphasized, since *CCMail* didn't provide this feature at all, but is now a main part of *CleverMail*. This will ensure that the im-


plementations will be encapsulated within *CleverMail* and can be used by other software components.

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 importiert und exportiert werden können.
- *CleverSupport* ist ein  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 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 ersetzen, 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 *CleverMail* 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 *CCMail*

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 möglichem 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.

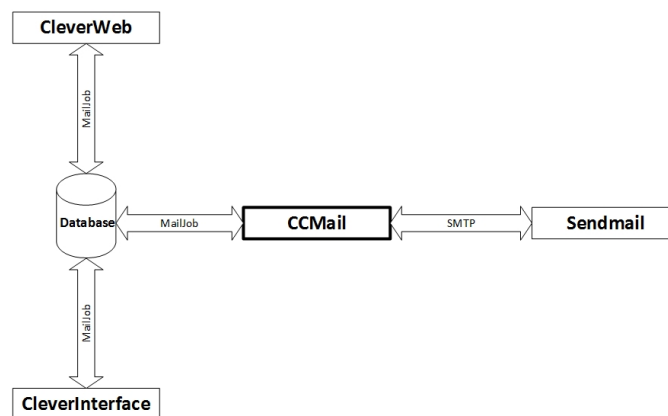





Abbildung 2.1: Systemaufbau und Integration von *CCMail*

Abbildung 2.1 zeigt das Gesamtsystem aus der Sicht der Anwendung *CCMail*, wobei anzumerken ist, dass das System bis heute angewachsen ist und nunmehr aus mehreren Anwendungen  wie in Abbildung 2.1 abgebildet besteht. Es lässt sich ableiten, dass  Kernstück des Systems die Datenbank  ist. In der Datenbank werden die zu versendenden *E-Mails* 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 [3, S. 27] treffend:

*The greater the coupling, the harder is to refactor 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*, welches 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* durch *CCMail*.

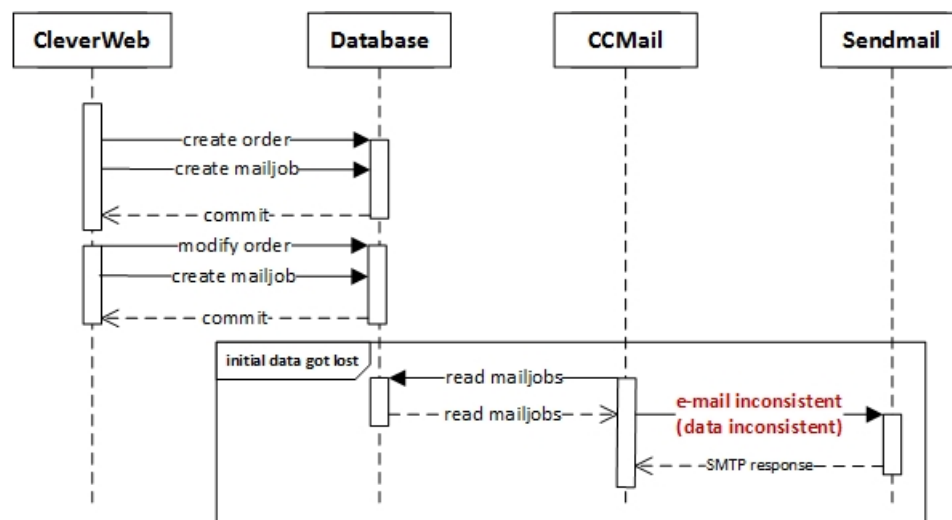


Abbildung 2.2: Gesamtprozess des *E-Mail*-Versands

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 *MailJob-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, dass 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 benötigt werden, zur Verfügung, wie:

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

Die Klasse *CCBasicEmail* implementiert die gesamte Funktionalität für den Versand einer *E-Mail* und ist die Basisklasse aller 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*.

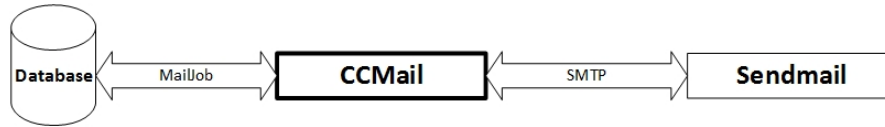


Abbildung 2.3: Teilsystem *CCMail*

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.

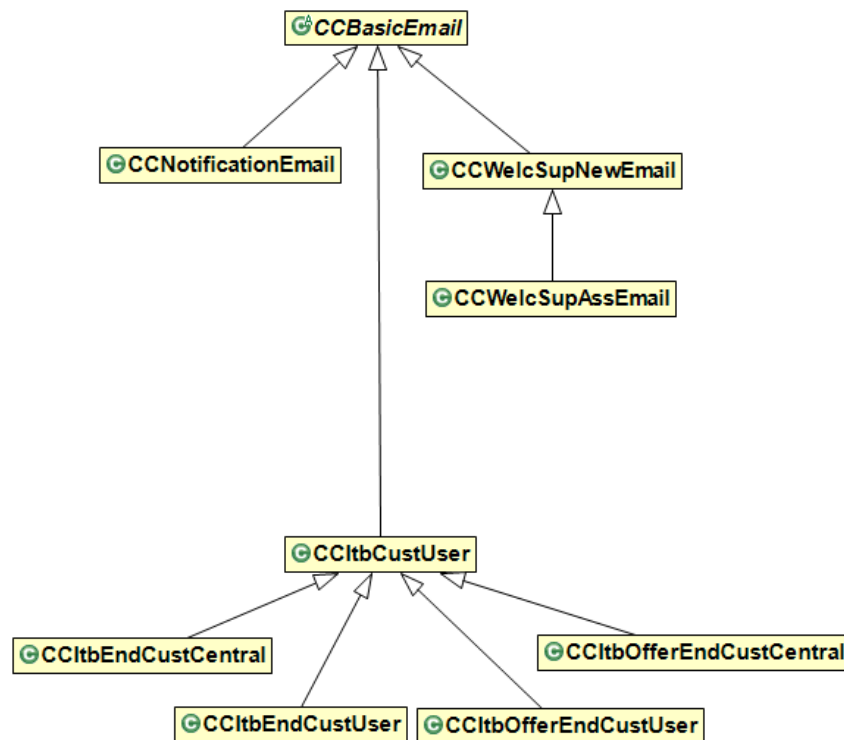



Abbildung 2.4: Auszug aus der Vererbungshierarchie von *CCBasicEmail*

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 Verer-

bungshierarchien 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  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 Parameter 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 entschieden 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* 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.

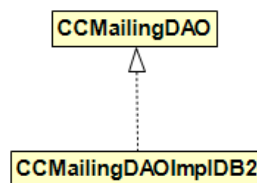


Abbildung 2.5: Vererbungshierarchie von *CCMailingDao*

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 Wartung 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.

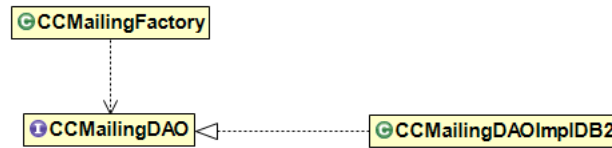
Eigene Schnittstellen und Implementierungen je *E-Mail*-Typ hätten es ermöglicht, für jeden dieser *E-Mail*-Typen Ressourcen zur Verfügung zu stellen, die nur dieser *E-Mail*-Typ verwendet. Mann hätte Flexibilität erhalten und hätte sich trotzdem auf eine gemeinsame Basis einigen können.

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* [2, 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

Abbildung 2.6: *CCMailingDaoFactory* für *CCMailingDao*

man auf diese *Factory* verzichten können, da die Abstraktion bereits über 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 *CCMail*-Datenbank

Abschließend wird der Aufbau des Datenbankschemas betrachtet, welches die Kernkomponente des Systems aus der Sicht von *CCMail* darstellt. Bei diesem Schema wurde auf Fremdschlüssel verzichtet, was grundsätzlich nur in Spezialfällen anzuraten ist.

Im Buch *Refactoring Database* [3, S. 213] wird als Argument für nicht verwendete Fremdschlüssel die Performanz genannt, wobei in diesem Fall diese Begründung nicht hält. 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 muss die Performance ohnehin kein Problem darstellen, da hier keine Vielzahl von TeilnehmerInnen und die keine Konkurrenz nicht geben ist. Das Problem von nicht verwendeten Fremdschlüsseln wird in *Refactoring Databases* [3, S. 213] wie folgt beschrieben:

The fundamental 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.

Es 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, ob dieser Ansatz ein guter ist?

Wie in Abbildung 2.7 ersichtlich, wurden die Spalten der Tabellen mit einem Präfix versehen, der eindeutig über das gesamte Datenbankschema ist. Mann sollte wissen, dass es ausreicht, dass die Spaltennamen eindeutig innerhalb des Kontexts einer Tabelle sind und nicht global über das gesamte Datenbankschema. Ebenso erkennt man, dass folgende Tabellen Fremdschlüssel definieren:

- *MAIL_JOB_ATTACHMENT_CONTAINERS*:
ist die Tabelle, welche die gebündelten Anhänge (*.zip) für einen *MailJob* repräsentiert.
- *MAIL_ATTACHMENT_CONTAINER*:
ist die Tabelle, welche ein Bündel von Anhängen für einen *MailJob* repräsentiert.
- *MAIL_ATTACHMENT_CONTAINER_ATTACHMENT*:
ist die Tabelle, welche einen Anhang eines *MailJob* repräsentiert.
- *MAIL_ATTACHMENT_PARTS*:
ist die Tabelle, welche die partitionierten und in *BASE64* kodierte Daten der Anhänge des *MailJob* repräsentiert.
- *MAIL_ATTACHMENT*:
ist die Tabelle, welche einen Anhang eines *MailJob* repräsentiert.
- *MAIL_JOB_ATTACHMENT*:
ist die Tabelle, welche die Anhänge eines *MailJob* repräsentiert.

Diese Tabellen wurden nachträglich hinzugefügt und man hat den Ansatz des Verzichts auf Fremdschlüssel offensichtlich aufgegeben. Diese Tabellen werden dazu verwendet, um Datei- Anhänge von *E-Mail* zu verwalten, die bereits bei der Erstellung des *MailJobs* vorhanden sind. Dies war eine Umgehungslösung und darf so auch nicht mehr angewandt werden, da hier die Dateien kodiert in Base64 (*Codepage* unabhängige ASCII- Zeichenfolge) gehalten werden und die Datenbank unnötig mit Daten belasten. Sie sollten in einem Dateisystem oder Dokumentenverwaltungssystem verwaltet und lediglich referenziert werden, was aber zum Zeitpunkt der Implementierung noch nicht zur Verfügung stand.

2.4.1 Datenbankschemata von *CleverMail*

Folgende Abbildung 2.7 illustriert das Datenbankschema von *CCMail*.

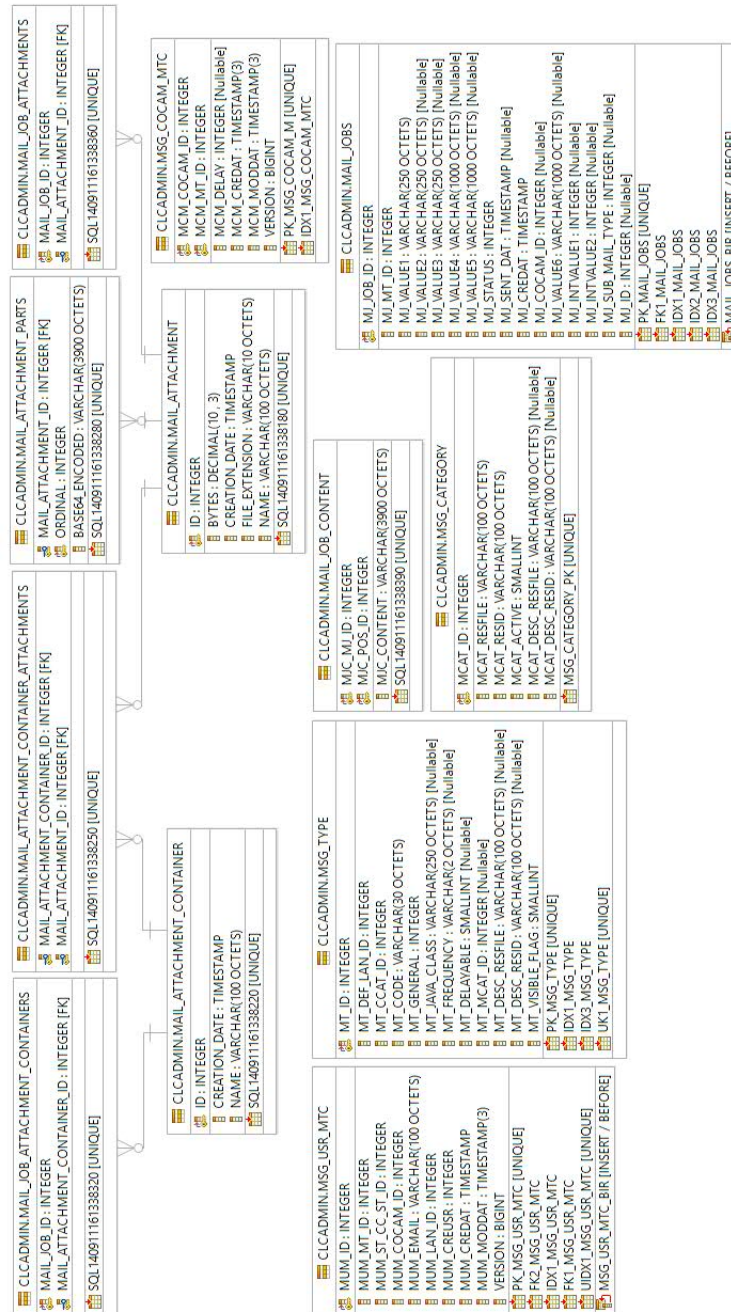


Abbildung 2.7: Datenbankschema *CCMail*

Mit den Betrachtungen des Datenbankschemas von *CCMail* ist die Analyse von *CCMail* abgeschlossen. Bei der Analyse wurden einige Probleme und Fehlentscheidungen von *CCMail* identifiziert wie z.B.:

- Vererbungshierarchie der *Mail*-Typen.
- Statische *Mail*-Vorlagen.
- Verzicht auf Datenbank-Fremdschlüssel.
- Global eindeutige Tabellenspaltennamen.

Die Ergebnisse der Betrachtungen stellen eine Grundlage für die Konzipierung von *CleverMail* dar. Diese Grundlage wird die Konzipierung von *CleverMail* erheblich erleichtern.

Kapitel 3

Die neue Anwendung *CleverMail*

In diesem Kapitel wird das Konzept von *CleverMail* erörtert, wobei *CleverMail* die bestehende Anwendung *CCMail* ablösen wird. Im Gegensatz zu *CCMail* wird aus der Sicht von *CleverMail* das Gesamtsystem aus mehreren Anwendungen bestehen, die in der Lage sein müssen *E-Mails* zu versenden. Bis heute ist *clevercure* angewachsen und es wurden neue Anwendungen hinzugefügt, die Aufgrund der Architektur von *CCMail* nicht in *CCMail* eingebunden werden konnten bzw. man sich dazu entschieden hat, die Einbindung zu unterlassen.

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

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

Im Gegensatz zu *CCMail* soll *CleverMail* nicht als Konsolen-Anwendung, sondern als eigenständige Komponente implementiert werden, die in einem Applikationsserver, der die JEE7-Plattform-Spezifikation unterstützt, betrieben werden kann.

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

1. *JAX-RS 2.0* (Java Api for RESTful Web Services 2.0),
2. *EJB 3.1* (Enterprise Java Bean. Standard Komponenten für die Entwicklung in Java Enterprise Containern),
3. *JPA 2.1* (Java Persistence Api. Java Schnittstelle für Datenbankzugriffe),
4. *JTA 1.2* (Java Transaction Api. Java Schnittstelle für den Support von verteilten Transaktionen),
5. *JSF 2.2* (Java Server Faces. Java Spezifikation für die Entwicklung von Webanwendungen) und
6. *CDI 1.2* (Context and Dependency Injection. Java Spezifikation eines IOC-Containers (Inversion of control container)).

Diese Bibliotheken 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 Beispiele für Enterprise-Anwendungen an, um zu illustrieren, dass jede dieser Anwendungen seine eigenen Probleme und Komplexität mit sich bringt. 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. als Unterstützung anzusehen sind und es keinen echten Leitfaden gibt, an dem man sich orientieren kann. Die Architektur wird stark von der konkreten Anwendung beeinflusst. Daher kann sich die Architektur je nach Anwendung stark unterscheiden.

3.1 Systemaufbau

Im Gegensatz zum Systemaufbau aus der Sicht von *CCMail*, beschrieben in Abbildung 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 die Datenbank von den Anwendungen abstrahiert werden. Damit erreicht man, dass die Anwendungen eine einheitliche Schnittstelle nutzen und nicht ihrerseits eigene Schnittstellen zur Datenbank implementieren und warten müssen.

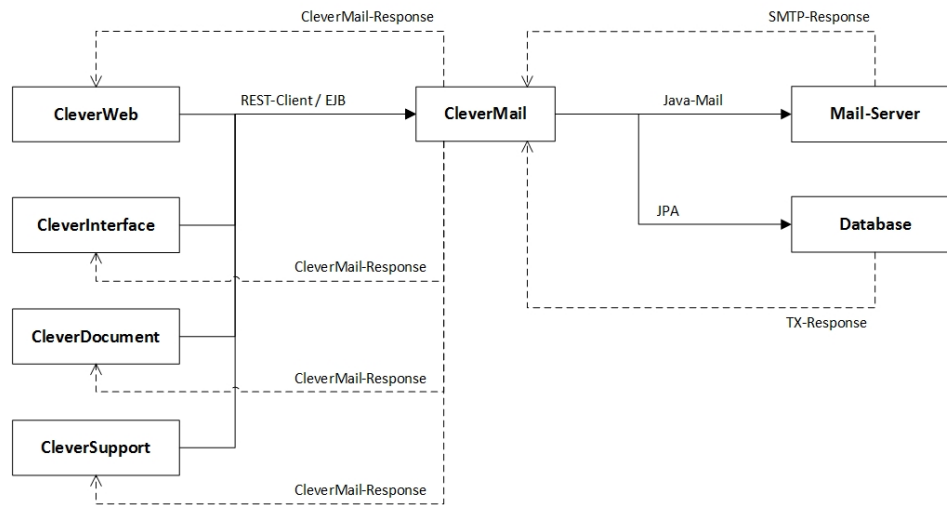


Abbildung 3.1: Systemaufbau und Integration von *CleverMail*

Wie in der Abbildung 3.1 illustriert, wird 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 einem *REST-Webservice*, über den die zur Verfügung gestellten Funktionalitäten genutzt werden können.
2. *EJB*, ein *EJB* (Enterprise Java Bean), welches die zur Verfügung gestellten Funktionalitäten bereitstellt.

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

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

1. *REST* nicht verfügbar,
2. *EJB* nicht verfügbar oder
3. Falsche Java-Version.

Obwohl diese Probleme auftreten können, kann zumindest gewährleistet werden, dass alle Anwendungen dieselbe Schnittstelle und dasselbe Domä-

nenmodell verwenden, selbst wenn eigene Implementierungen erforderlich sind. Diese Implementierungen würden Softwarekomponenten von *CleverMail* darstellen 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 Schnittstellen.

3.1.1 1. Möglichkeit (*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 den Anwendungen erfordern, die den *REST-Client* verwenden. Dem steht aber gegenüber, dass *REST-Services* zustandslos sind und sich daher nicht in Datenbank-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. beim Anlegen einer Bestellung). Für einen *REST-Service* startet der Lebenszyklus mit dessen Aufruf und endet mit dem Übermitteln der Antwort 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).

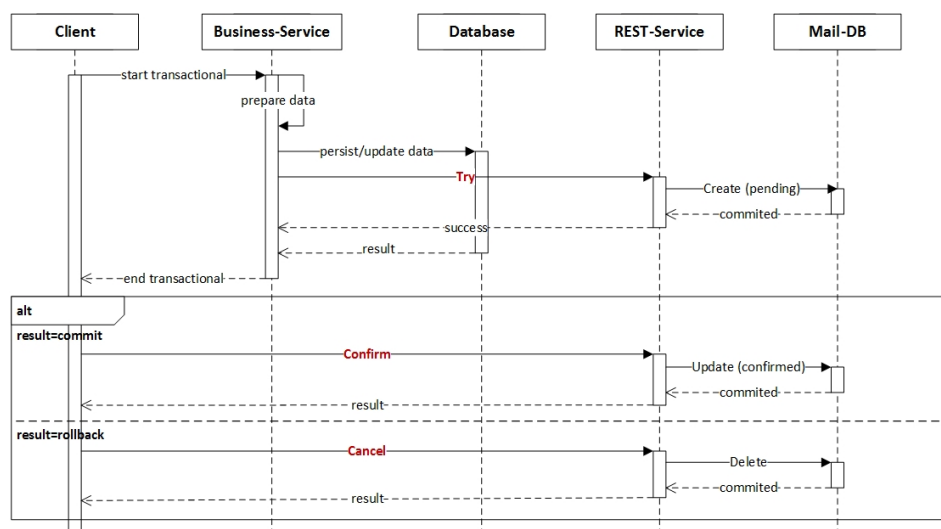


Abbildung 3.2: Beispiel einer Transaktion mit TCC

Mit dem Konzept *TCC* hält der *REST-Service* die *E-Mail* persistent, mar-

kiert die *E-Mail* aber als *unconfirmed*, damit diese *E-Mail* in keiner Verarbeitungslogik miteinbezogen wird. Nach dem erfolgreichen Abschluss der Transaktion auf der *Client*-Seite bestätigt der *Client* den durch den *REST-Service* persistent gehaltenen Zustand, und im Falle eines Fehlers erklärt der *Client* den Zustand für ungültig. Dies erfordert zwei Aufrufe zu *REST-Services*. Ebenfalls sollte die Transaktion über einen Transaktionskoordinator auf der *REST*-Seite kontrolliert werden, was wiederum einen Mehraufwand bedeutet. Dieser Transaktionskoordinator wäre dafür verantwortlich, die *REST-Services*, die Teil einer logischen Transaktionen sind, zu managen.

Mit *TCC* besteht auch die Gefahr das eine *Heuristic-Exception* auftritt. Eine *Heuristic-Exception* tritt auf wenn ein Teilnehmer der Transaktion eine *Heuristic-Decision* (eigenmächtige Entscheidung) trifft und dadurch Dateninkonsistenzen auf der Datenbank entstehen. Dies ist ein Problem, dass vor allem in verteilten System auftreten kann.

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 kann die Erreichbarkeit des *REST-Service* sein. Sollte dieser einmal ausfallen, oder im Falle eines Neustarts nicht erreichbar sein, so müsste man eine Rückversicherung haben und die zu erstellenden *E-Mails* anderweitig zwischenspeichern, wie z.B. in Form einer Textdatei, welche die Daten in Form von JSON (Javascript-Objekt-Notation) enthält.

Der Artikel [4] beschreibt den Prozess von *TCC* mit mehreren involvierten *REST-Services* gut und detailliert.

3.1.2 2. Möglichkeit (*EJB*)

Sollte eine Anwendung *CleverMail* via dessen zur Verfügung gestellten *EJBs* verwenden, so würde eine starke Kopplung und starke Abhängigkeiten entstehen, da mehr Ressourcen benötigt werden. Ebenso könnte im Gegensatz zu einem *REST-Client* keine eigene Datenbank genutzt werden, da ein Zweiphasen-*Commit* erfolgen müsste. Natürlich würde die Möglichkeit von mehreren verwendeten Datenbanken bestehen, aber man wäre der Gefahr von *Heuristic-Exceptions* ausgesetzt.

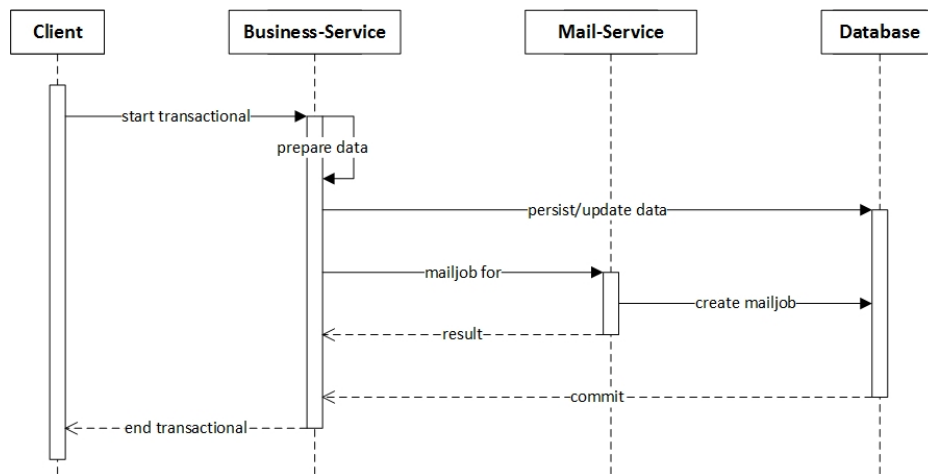


Abbildung 3.3: Beispiel einer Datenbanktransaktion mit *EJB*

Wie in Abbildung 3.3 illustriert erfolgt das Anlegen einer *E-Mail* in derselben Transaktion und würde daher auch im Falle eines *Rollbacks* entfernt werden. Dies ist sicher die einfachste Art und Weise, um *E-Mails* anzulegen, da hier keine besonderen Mechanismen implementiert werden müssen um die Datenkonsistenz zu gewährleisten. Mit diesem Ansatz wäre die *E-Mail* Teil einer wohldefinierten Transaktion.

Wie im Kapitel 3 angemerkt, können technologische Probleme auftreten, wenn z.B. die Laufzeitumgebung einer Anwendung *EJB* und *JTA* nicht zur Verfügung stellt. In so einem Fall müsste man eigene Implementierungen zur Verfügung stellen, die Teil von *CleverMail* sein müssen.

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

3.2 *E-Mail*-Prozesse

Dieser Abschnitt behandelt die Prozessspezifikationen von *CleverMail*. Es wird das Augenmerk auf den Mailversand gelegt. Grundlegend wird sich der *E-Mail*-Versand dadurch unterscheiden, dass mehrere Ebenen involviert sind, bevor eine *E-Mail* 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 in-

tern konfigurierbar sein sollen, sondern ebenfalls durch die KundInnen selbst konfiguriert und gesteuert werden können. Es sollen folgende Konfigurationsmöglichkeiten zur Verfügung stehen.

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

Zur Zeit stehen diese Möglichkeiten, wenn vorhanden, nur intern zur Verfügung. Die KundInnen haben lediglich die Möglichkeit, einzelne *E-Mail*-Typen zu aktivieren oder zu de-aktivieren. Diese *E-Mail*-Typen können aber mehrere *E-Mails* halten. Das grundlegende Ziel ist, dass die KundInnen 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. Trotz etwaiger Einschränkungen, sollen die KundInnen in der Lage sein, den *E-Mail*-Verkehr ihrer *E-Mails* besser zu steuern.

3.2.1 *E-Mail*-Versand

Folgende Abbildung 3.4 illustriert ein Beispiel eines *E-Mail*-Versands, wobei ein *Client* über eine *REST-Service*-Schnittstelle den Versand von *E-Mails* eines Typs anstößt. Dabei wird der vollständige Prozess eines *E-Mail*-Versands, wie angedacht, dargestellt.

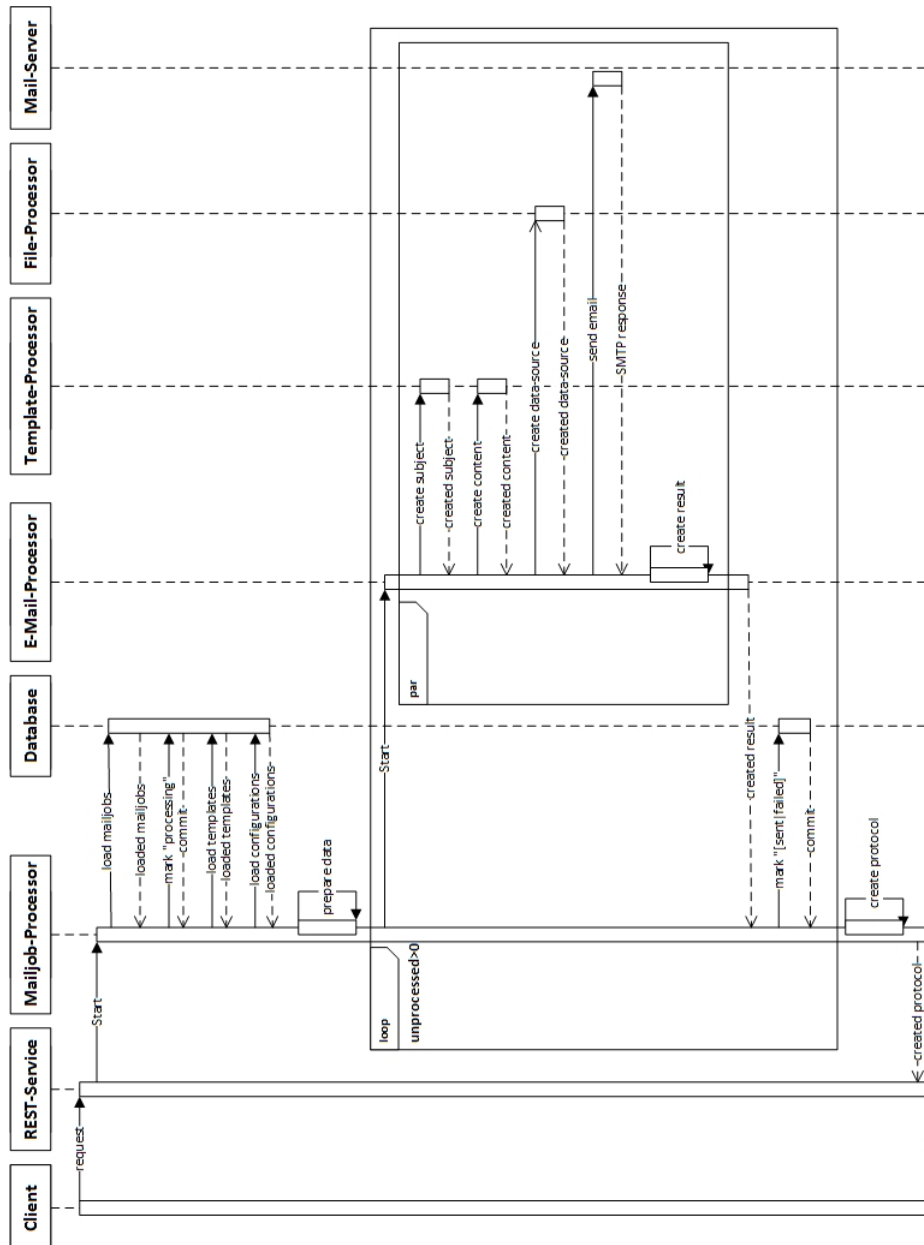


Abbildung 3.4: Prozess des *E-Mail*-Versands

Der in Abbildung 3.4 dargestellte Prozess wird über eine *REST-Service*-Schnittstelle angestoßen, der in diesem Fall den Prozess synchron abarbeitet und anschließend eine Antwort in Form eines erstellten Reports zurückliefert. Der *REST-Service* ist 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.


Im folgenden sei der Prozess in Unterpunkte unterteilt angeführt und beschrieben:

1. *MailJob*-Aufbereitung

Nachdem der Prozess des *E-Mail*-Versands gestartet wurde, müssen die zu verarbeitenden *E-Mails*, repräsentiert durch *MailJobs* aus der Datenbank geladen und mit *Processing* markiert werden. Im Abschnitt 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 mit dem Ansatz des Markierens der *MailJob*-Entitäten nicht mehr. Nachdem die *MailJobs* geladen wurden, müssen die zu den *E-Mail*-Typ dazugehörigen *E-Mail*-Vorlagen geladen werden. Es würde sich ein *Cache*-Mechanismus anbieten, da die *E-Mail*-Vorlagen versioniert sind und sich die *E-Mail*-Vorlagen einer Version nicht mehr ändern dürfen. Anschließend werden die kundenspezifischen Konfigurationen geladen. Aus diesen Daten werden Objekte erstellt, die eine *E-Mail* repräsentieren. Diese Modelle werden in weitere Folge dazu verwendet, die tatsächlichen *E-Mails* zu erstellen.

2. *E-Mail*-Erstellung

Aus den erstellten Modellen werden die *E-Mails* erstellt. Dabei wird dieser dreiteilige Prozess angewendet:

1. *Erstellen des Betreff*. Der Betreff wird aus einer *E-Mail*-Vorlage erstellt und mit Daten befüllt, wenn in der Vorlage Parameter definiert wurden.
2. *Erstellen der Nachricht*. Die Nachricht wird aus einer *E-Mail*-Vorlage erstellt und mit Daten befüllt, wenn in der Vorlage Parameter definiert wurden.
3. *Erstellen der DataSources*. *DataSource*-Objekte halten die Anhänge der *E-Mails*. Die stellen ein  *Stream* über die Anhänge der *E-Mail* zur Verfügung.

Die Verarbeitung der *E-Mail*-Vorlagen werden in einer Komponente mit dem

Namen *Template-Processor* erfolgen. Nachdem die Werte der verwendeten Vorlagenparameter beim Erstellen eines *MailJobs* in der Datenbank gespeichert wurden, können diese angewandt werden.

Die Verarbeitung der Datei Anhänge erfolgt in einer Komponente mit dem Namen *File-Processor*, welche die verlinkten Dateianhänge in Form von *DataSource*-Objekten dem *E-Mail*-Objekt zur Verfügung stellt. Beim Versand würde die das *E-Mail*-Objekt den Dateianhang über die verfügbaren *DataSource*-Objekte laden. Dabei werden die Dateien in Form von Links aus dem Dokumentenmanagementsystem *CleverDocument* geladen. Es dürfen keine Dateien in Form von Base64-Zeichenketten in der Datenbank gespeichert werden, damit die Datenbank nicht mit unnötigen Daten belastet wird. Es könnten hierbei verschiedene *DataSource*-Implementierungen zur Verfügung gestellt werden, welche die Dateien aus verschiedenen Quellen über verschiedene Protokolle laden können (z.B. REST, SOAP oder HTTP).

3. *E-Mail*-Versand

Es sollte angedacht werden den *E-Mail*-Versand sowie die Erstellung der *E-Mail*-Objekte asynchron erfolgen zu lassen, da eine sequenzielle Verarbeitung schlecht für die Performance wäre. Nach dem erfolgreichen oder fehlgeschlagenen Versand einer *E-Mail* werden die verfügbaren Daten des *E-Mail*-Versands in Form eines aufbereiteten Resultates zurückgeliefert. Dabei ist vor allem der *SMTP*-Status relevant, der auf jeden Fall gespeichert werden muss. Damit wird sichergestellt, dass die KundInnen nachvollziehen können, wie der Versand einer *E-Mail* abgearbeitet wurde und mit welchen *Error*-Code in einem Fehlerfall eine *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. *Mail-Server* nimmt Nachricht nicht an.

Es kann viele Ursachen haben warum eine *E-Mail* nicht zugestellt werden konnte. Vor allem die *Mail-Server* der KundInnen stellen hierbei einen Unsicherheitsfaktor dar, da diese in unterschiedlicher Art und Weise konfiguriert sein könnten. Diese Konfigurationen könnten dazu führen, dass *E-Mails* nicht beim Empfänger ankommen.

3.2.2 *E-Mail*-Vorlagen und -Parameter

Einen weiteren wichtigen Aspekt stellen die *E-Mail*-Vorlagen, deren Parameter und die Verwaltung derer dar. Die Bereitstellung von Vorlagenparametern wird den AnwenderInnen die Möglichkeit bieten, auf kontextabhängige Daten in einer *E-Mail*-Vorlage zugreifen zu können. Dadurch wird die Flexibilität erhöht und die AnwenderInnen werden mehr Freiheiten beim Erstellen einer *E-Mail*-Vorlage bekommen.

Die Vorlagenparameter müssen von den EntwicklerInnen innerhalb eines Kontexts zur Verfügung gestellt werden, wobei hier auch ein Entwicklungsaufwand besteht. Die Vorlagenparameter werden innerhalb deren Kontext evaluiert und in die *E-Mail*-Vorlage eingefügt. Dabei können die Daten aus verschiedenen Objekten oder sogar Objektgraphen kommen. Dies bedeutet, dass die Vorlagenparameter über eine spezielle Implementierung ausgelesen werden müssen, die leicht wartbar gehalten werden muss.

Die Vorlagenparameter werden über mehrere Softwarekomponenten hinweg verwendet, die sich in verschiedenen Laufzeitumgebungen befinden dürfen. Daher wird es erforderlich sein, die Vorlagenparameter in einem übergreifenden Modell zu definieren oder über sie von einem Modell in ein anderes Modell über Zuordnungen wie z.B. *Annotations* zu definieren.

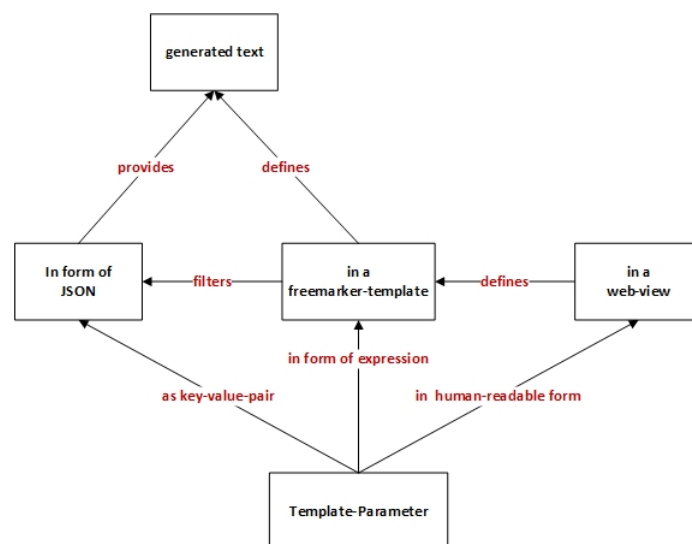


Abbildung 3.5: Verwendung von Vorlagenparametern

Wie in der Abbildung 3.5 illustriert, werden die Vorlagenparameter in verschiedenen Verwendungskontext und Darstellungen verwendet. Dadurch stellt sich die Frage, wie diese Vorlagenparameter adressiert werden. Man könnte

eine Zuordnung in jedem Verwendungskontext erstellen, also jeder Verwendungskontext bekommt sein eigenes Modell. Dadurch würde man zwar lose and die Vorlagenparameter gekoppelt sein , jedoch erhält man auch eine Modell-Klasse je Verwendungskontext, die gewartet werden muss.

Verwendungskontext Webseite

Da die Vorlagenparameter in *E-Mail*-Vorlagen verwendet werden, wird es auch eine Webseite geben müssen, über die die AnwenderInnen die Vorlagenparameter in Ihren *E-Mail*-Vorlagen verwenden können. Die Vorlagenparameter müssen für die AnwenderInnen einen Namen bekommen, der auch in mehreren Sprachen zur Verfügung stehen muss. Dadurch werden die Vorlagenparameter auf einen Schlüssel zugeordnet, der wiederum auf einen lokalisierten Spracheintrag zugeordnet ist. Die Zuweisung auf diesen Schlüssel ist erforderlich, aber eine Abstraktion der Vorlagenparameter für die Webseite bzw. deren *Backend* wäre übertrieben.

Verwendungskontext *Freemarker*-Vorlagen

Ein weiterer Aspekt ist auch die direkte Verwendung der Vorlagenparameter in der *Freemarker*-Vorlage selbst. Als *Template-Engine* wird für die Erstellung der *E-Mails* aus den *E-Mail*-Vorlagen *Freemarker* (Frei verfügbare *Template-Engine* in Java für die Erstellung von dynamischen Textdateien) verwendet. Diese Bibliothek ist eine beliebte und gut gewartete Bibliothek, welche alle benötigten Kontrollstrukturen sowie *Freemarker Expressions* zur Verfügung stellt. Diese *Freemarker Expressions* ähneln den *Java EL Expressions* (*Java Expression Language* ist eine Java-Spezifikation für *Expressions*). Die Vorlagenparameter werden in den *Freemarker Expressions* verwendet, die es ermöglichen, eine flache Adressierung sowie auch eine Adressierung über Objektgraphen zu definieren. Auch im Verwendungskontext der *Freemarker Expressions* ist eine Notwendigkeit einer Abstraktion der Vorlagenparametern in Frage zu stellen.

Verwendungskontext JSON

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

allen involvierten Komponenten verwendet werden.

3.3 *CleverMail*-Datenbank

Zusätzlich zur Persistenz der *E-Mails* soll auch die Möglichkeit bestehen, den *E-Mail*-Versand zu konfigurieren. Dies soll einerseits innerhalb des Betriebes möglich sein und andererseits auch für die AnwenderInnen der KundInnen. Dabei sollen folgende Möglichkeiten zur Verfügung stehen:

1. Zeitsteuerung des Versands.
2. Berechtigungen für Modifikationen der *E-Mail*-Vorlagen steuern.
3. Eigene *E-Mail*-Typen definieren.
4. Eigene *E-Mail*-Vorlagen definieren.
5. Zusatzdokumente für *E-Mails* definieren.
6. Speicherverwaltung der *E-Mails* steuern.

Um diese Konfigurationen zu verwalten wird ein Speichermedium benötigt, welches auch in der Lage sein muss, die nötigen Relationen abzubilden. Da bietet sich eine relationale Datenbank an. Auch in der alten Anwendung *CCMail* wurde eine relationale Datenbank verwendet und dies soll auch für die neue Anwendung *CleverMail* gelten. Dabei kann das bestehende Datenbankschema von *CCMail* außer Acht gelassen werden und ein vollständig neues Datenbankschema konzipiert werden.

Im Gegensatz zum Datenbankschema von *CCMail*, beschrieben im Abschnitt 2.4, darf keineswegs auf die referenzielle Integrität zwischen den Tabellen-Relationen verzichtet werden. Dies stellt die größte Fehlentscheidung beim Design des Datenbankschemas von *CCMail* dar. Des weiteren soll so gut wie möglich auf die Unabhängigkeit des Datenbankschemata geachtet werden. Damit ist gemeint, dass die Tabellen des Datenbankschemas von *CleverMail* nicht Tabellen der Anwendung *Clevercure* referenzieren dürfen. Dadurch wird sichergestellt, dass das Datenbankschema von *CleverMail* auch ohne *Clevercure* verwendet werden kann. Sollten Referenzen auf Tabellen des Datenbankschemas von *Clevercure* wie Benutzer (*USER*), Kunde (*COMPANY*) eingeführt werden, so kann *CleverMail* nicht außerhalb des Kontextes von *clevercure* existieren und würde immer nur für *Clevercure* zur Verfügung stehen. Das Modul *CleverMail* sollte aber auch als *Standalone*-Anwendung funktionsfähig sein, auch wenn es für die Anwendung *Clevercure* entwickelt werden soll.

Dies begründet sich durch die Anwendungsfälle selbst, die nicht auf eine bestimmte Repräsentation von Inhabern von persistenten *E-Mail*, *E-Mail*-Typen oder Konfigurationen angewiesen sind. Mann muss sich hier die Mög-

lichkeiten für eine zukünftige Anwendung in anderen Anwendungen offen zu halten, anstatt sich den Aufwand einer umfangreichen Umstrukturierung, beim Eintreffen eines solchen Falles, auszusetzen.

Auch wenn der Ansatz der größtmöglichen Abstraktion von *Clevercure* verfolgt werden soll, ist es trotzdem möglich Referenzen auf Tabellen des Datenbankschemas von *Clevercure* herzustellen, solange der Datenzugriff gekapselt und nur einer Stelle gehalten wird. Somit ist gewährleistet, dass eine Umstrukturierung keine unerwünschten Nebeneffekte hat und Änderungen nur an definierten Stellen stattfinden müssen. Scott W. Ambler und Parmond J. Sadalage beschreiben es in *Refactoring Databases* [3, 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 implement the data access logic separately from business classes.

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

Das Datenbankschema aus Abbildung 3.6 enthält keine Referenzen auf das Datenbankschema von *Clevercure*. Die Integration des Datenbankschema von *CleverMail* in das Datenbankschema von *Clevercure* oder visa versa, 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 Fremdschlüssel der referenzierten Tabelle hinzugefügt werden muss. Man erhält 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 aus Abbildung 3.6 von *CleverMail* sehr dem Datenbankschemata 2.7 ähnelt. Wobei neue Tabellen zu dem Datenbankschema von *CleverMail* hinzugefügt wurden, um die neuen Möglichkeiten wie die Steuerbarkeit des *E-Mail*-Versandes zu ermöglichen.

Die Tabellen, welche die Konfigurationen halten, sollten je nach Anforderung umgesetzt werden, wobei die Möglichkeit Konfigurationen zur Verfügung zu stellen, stets möglich sein soll. Dadurch werden auch die AnwenderInnen der KundInnen nicht unnötig mit nicht benötigten Konfigurationsmöglichkeiten belastet.

Kapitel 4

Zusammenfassung

Zusammenfassend kann Ich sagen, dass das Ausarbeiten dieser Bachelorarbeit sich zeitweise als Herausforderung herausgestellt hat. Das Verfassen einer wissenschaftlichen Arbeit und die damit einhergehenden Vorschriften waren anfänglich ungewohnt für mich. Aber schlussendlich sehe Ich auch, dass diese Vorschriften und Konventionen durchaus ihren Sinn haben. Das Resultat ist eine gut strukturierte wissenschaftliche Arbeit, die dem Leser ein Themengebiet auf wissenschaftliche Art und weise näher bringt.

Ich habe mich für das Thema "Konzeption eines *Mail-Service* entschieden, da mir folgende Punkte wichtig waren:



1. Das Design und die Architektur der alten Anwendung *CCMail* zu analysieren.
2. Das Konzept für eine neue Anwendung *CleverMail* zu erstellen.
3. Neue Möglichkeiten und *Framworks* kennen zu lernen.

Vor allem die Architektur der neuen Anwendung *CleverMail*, deren Schichten sowie mit ihr interagierende Softwarekomponenten erschien mir wichtig. Diese Annahme stellten sich für mich als richtig heraus, da bei der Analyse von *CCMail* herausgestellt hat, dass die Interaktion mit anderen Softwarekomponenten nicht vorgesehen wurde. Dies ist vor allem in der Verwendung der Datenbank als Schnittstelle begründet. Dadurch ist *CCMail* nicht vollständig in das Gesamtsystem von *Clevercure* integriert.

Das Konzept von *CleverMail* ist flexibel genug um mit anderen Softwarekomponenten interagieren zu können. Das wurde durch folgende Schnittstellen ermöglicht:

1. REST-Service.
2. *EJB* oder *DAO*

Durch diese Schnittstellen wird die Datenbank als Schnittstelle abgelöst und wird damit von den Anwendungen abstrahiert. Das stellte für mich einen schweren Designfehler dar, da eine Kopplung zwischen Softwarekomponenten über eine Datenbank nicht zu empfehlen ist. Man hat es hier versäumt die nötige Abstraktion zwischen *CCMail* und den Anwendungen wie z.B. *CleverWeb* einzuhalten.

Für den weiteren Verlauf sehe ich die größte Herausforderung in den Vorlagenparametern und dessen Verwendungskontexte. Hier wird man ein hohes Maß an Konsistenz einhalten müssen, um Probleme zu vermeiden. Die weit gestreute Verwendung der Vorlagenparameter über die verschiedenen Verwendungskontexte wie:

- Webseite oder
- *Freemarker*-Vorlage

werden Umstrukturierungen erschweren. Auch die Handhabung der Vorlagenparameter durch die Anwenderinnen über die Webseite ist schwierig, da es hier keinen etablierten Ansatz gibt, mit dem man dieses Problem lösen könnte. Es wird viel Eigenarbeit erfordern, das umzusetzen. Trotz dieser Herausforderungen wird sich das Konzept von *CleverMail* ohne größer Schwierigkeiten umsetzen lassen. Alle angedachten Bibliotheken sind entweder in einem Standard spezifiziert oder haben sich über die Zeit etabliert und werden von der Entwicklergemeinde anerkannt.

Quellenverzeichnis

Literatur

- [1] Martin Fowler. *Patterns of Enterprise Application Architecture*. 1. Aufl. München: Addison-Wesley Professional, 2002 (siehe S. 19).
- [2] Joshua Kerievsky. *Refactoring to Patterns*. München: Addison-Wesley Professional, 2004 (siehe S. 13).
- [3] Scott W.Ambler und Pramond J.Sadalage. *Refactoring Databases*. München: Addison-Wesley Professional, 2006 (siehe S. 5, 14, 31).

Online-Quellen

- [4] Atomikos Corporate Headquarters. *Transactions for the REST of us*. 2012. URL: [http : / / www . atomikos . com / Publications / TransactionsForTheRestOfUs](http://www.atomikos.com/Publications/TransactionsForTheRestOfUs) (siehe S. 22).