file:///B:/Google Drive/FH-Hagenaberg/Bachelorarbeit/Burger/hgb-thesis-utf-20141105/hgbthesis.cls

Konzeption eines Mail Service

Ing. Thomas Herzog

BACHELORARBEIT

Nr. XXXXXXXXXX-B

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2016

Diese Arbeit entstand im Rahmen des Gegenstands

Software Engineering

im

Sommersemester 2016

Betreuer:

FH-Prof. DI Dr. Heinz Dobler

Erklärung

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

Hagenberg, am 1. Februar 2016

Ing. Thomas Herzog

Inhaltsverzeichnis

Vorwort

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

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

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

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

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

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

Kurzfassung

// TODO: Zusammenfassung nach Abschluss der Arbeit in Deutsch

Abstract

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

Kapitel 1

Einleitung

1.1 Zielsetzung

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

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

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

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

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

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

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

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

Kapitel 2

CCMail

In diesem Kapitel wird die existierende Anwendung *CCMail* diskutiert und analysiert. Ziel ist es einen Überblick über diese Anwendung und dessen Aspekte zu liefern sowie diese Aspekte genauer zu betrachten und zu analysieren. Die Ergebnisse dieser Analyse sollen als Grundlage für das neue Konzept dienen, das auch das bestehende System bzw. dessen Aufbau und Implementierungen berücksichtigen muss. Dies trifft vor allem auf die Anwendungen *CleverWeb* und *CleverInterface* zu, die die Neuentwicklung *CleverMail* integrieren müssen. Diese Integration soll natürlich mit geringstem Aufwand erfolgen können, da hier Probleme bei der Integration negative Auswirkungen auf den produktiven Betrieb der Firma *curecomp* haben könnten.

2.1 Systemaufbau

Im folgenden wird der Systemaufbau aus der Sicht der Anwendung *CCMail* und dessen Integration in dasselbe diskutiert.

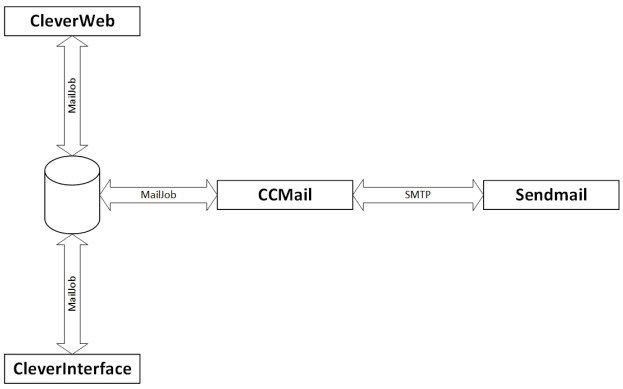

Abbildung 2.1: Systemaufbau und Integration von *CCMail*

Dieses Diagramm zeigt das Gesamtsystem aus der Sicht der Anwendung *CCMail* wobei anzumerken ist dass das System bis heute angewachsen ist und nunmehr aus mehreren Teilsystem besteht. Es lässt sich hier 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*, der alle Informationen einer E-Mail-Nachricht enthält.

Die Anwendungen *CleverWeb* und *CleverInterface* erstellen über ihre eigens implementierte Datenbankzugriffsschicht *MailJob* Entitäten in der Datenbank, welche zeitgesteuert von *CCMail* ausgelesen, verarbeitet und versendet werden. *CCMail* ist als eine Konsolen-Anwendung Implementiert und enthält alle Ressourcen, die es benötigt um die *MailJob* Entitäten zu verarbeiten.

Als Mail-Server wird *Sendmail* verwendet. Es handelt sich hierbei um eine Anwendung, die für Linux Distributionen frei verfügbar ist. *CCMail* versendet über das SMTP-Protokoll die E-Mail-Nachrichten an die *Sendmail* Instanz, die sie ihrerseits den Empfängern zur Verfügung stellt.

2.1.1 Gesamtprozess

Der folgend beschriebene Gesamtprozess soll aufzeigen wie in dem vorherig beschriebenen System der E-Mail-Versand vom Anlegen eines *MailJob* bis hin zum Versand der eigentlichen E-Mail-Nachricht funktioniert. Als Kernkomponente des Systems wurde die Datenbank identifiziert, die die *MailJob* Entitäten hält, die wiederum von *CCMail* 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-Nachrichten, durch die Zeitdifferenz zwischen dem Anlegen eines *MailJob* durch die angeschlossenen Anwendungen und dem tatsächlichen Versand der E-Mail-Nachricht durch *CCMail*.

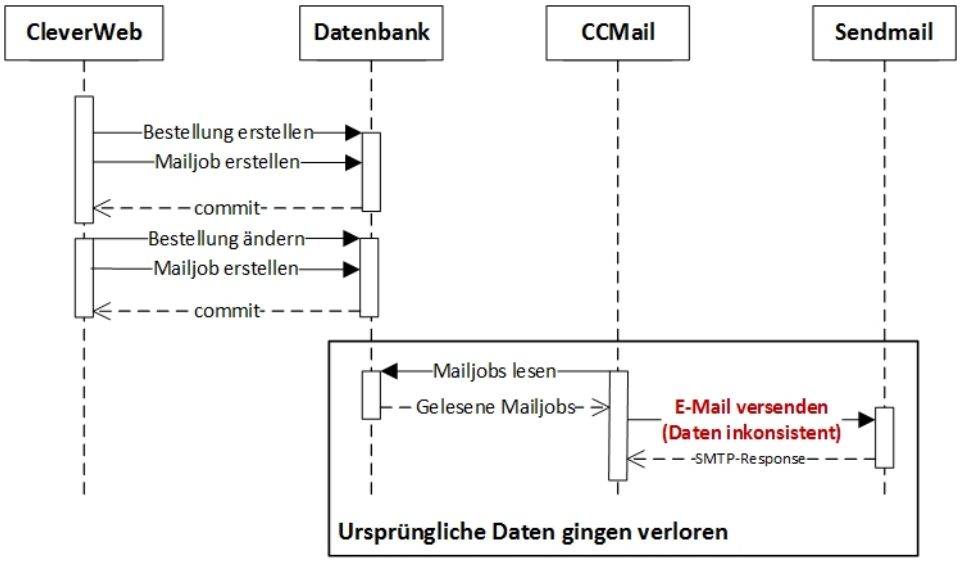

Abbildung 2.2: Beispiel für den Gesamtprozess eines E-Mail Versandes

Wie man aus dem Sequenz Diagramm ableiten kann ist eines der Hauptprobleme an diesem Prozess ist die Inkonsistenz, was in der Art und Weise wie die *MailJob* Entitäten verarbeitet werden begründet ist. Aufgrund der zeitgesteuerten bzw. zeitversetzten Verarbeitung kann es vorkommen das Daten einer E-Mail-Nachricht sich ändern bevor diese überhaupt versendet wurde. In diesem Beispiel wird eine Bestellung angelegt und kurz darauf geändert bevor die E-Mail-Nachricht über die Erstellung der Bestellung versendet wurde. Es wurde zwar ein neuer *MailJob* angelegt aber beide *MailJob* Einträge verweisen auf dieselbe Bestellung. Dadurch enthalten beide versendeten E-Mail-Nachrichten dieselben Informationen und die Informationen der ursprünglich erstellten Bestellung gingen verloren.

Dies ist begründet in der Art und Weise wie die *MailJob* Einträge aufgebaut

sind. Ein *MailJob* hält die Informationen für den Versand einer E-Mail wobei hierbei nicht die gesamte E-Mail-Nachricht oder die verwendeten Daten gespeichert werden sondern lediglich die Parameter, die in einer SQL-Abfrage benutzt werden um die Daten für die E-Mail-Nachricht 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-Nachricht nicht mehr möglich bzw. e kann nicht garantiert werden, dass diese Nachricht dieselben Informationen enthält wie beim ersten Versand.

Ein Weiteres Problem liegt in der zeitgesteuerten Verarbeitung der *MailJob* durch *CCMail*. Es wurde lange Zeit nicht geprüft ob bereits ein *CCMail* Prozess gestartet wurde bevor dieser gestartet wird. Dies hat dazu geführt dass es vorkam das mehrere Prozesse gleichzeitig die *MailJob* Entitäten verarbeiten und daher die E-Mail-Nachrichten mehrmals versendet wurden. Dieses Problem ist begründet durch die Tatsache dass in Verarbeitung stehende *MailoJob* Entitäten nicht als solche markiert wurden und von dem parallel laufende Prozess ebenfalls ausgelesen und verarbeitet wurden. Zurzeit wird lediglich geprüft ob bereits ein Prozess gestartet wurde, was es unmöglich macht die Arbeit auf mehrere Prozesse aufzuteilen.

2.2 Design

Nachdem der Systemaufbau diskutiert wurde wird sich nun mit dem Design von *CCMail* befasst. *CCMail* wurde als Konsolen-Anwendung implementiert und hält alle implementierten E-Mail-Typen, die zur Verfügung stehen wie:

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

Als Kernpunkt der Anwendung bzw. dessen Implementierungen ist eine die Klasse namens *CCBasicEmail* anzusehen, die die gesamte Funktionalität für den Versand einer E-Mail-Nachricht enthält, sowie eine Klasse namens *CCMailingDao* die alle Datenbankabfragen über alle E-Mail-Typen hinweg enthält. Dieser Teil soll die Schwächen der bestehenden Implementierung und

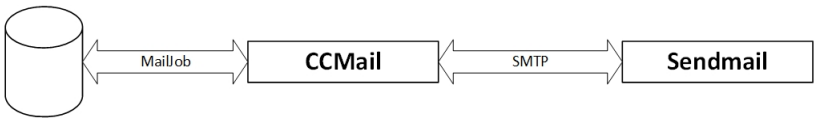

Abbildung 2.3: Teilsystem *CCMail*

dessen Design betrachten und analysieren. Die Ergebnisse dieser Betrachtungen und Analyse sollen bei der Erstellung des neuen Konzeptes mit einfließen und verhindern dass bereits gemachte Fehlentscheidungen sich wiederholen, sowie auch mögliche gute Ansätze weiterverfolgt werden.

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

1. *CCBasicEmail*
   Wurzelklasse aller E-Mail-Typen, die als abgeleitete Klassen von *CCBasicEmail* implementiert wurden
2. *CCMailingDao*
   Schnittstelle zur Datenbank, welche alle Abfragen auf die Datenbank über alle E-Mail-Typen hinweg enthält.
3. *CCMailingFactory*
   Diese Klasse ist für das Erstellen von *CCMailingDao* Instanzen verantwortlich

2.2.1 CCBasicEmail

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

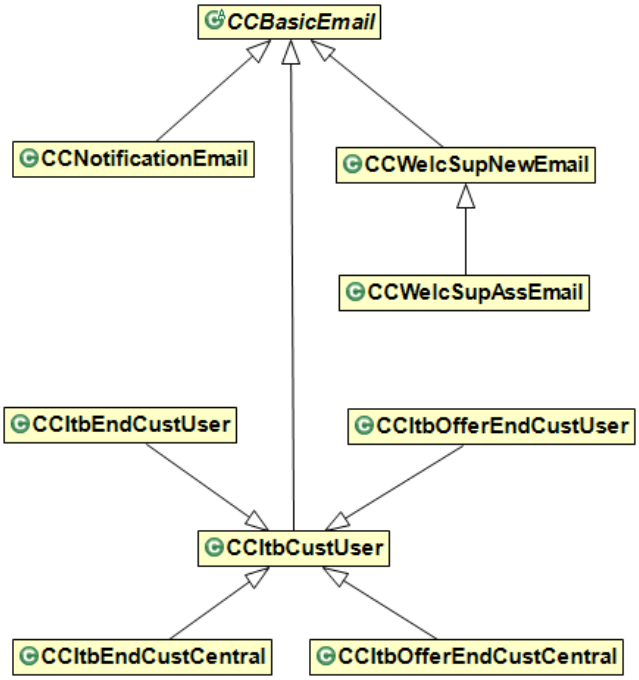

Abbildung 2.4: Auszug aus der Klassenhierarchie von *CCBasicEmail*

gramm lässt sich ableiten dass man sich dazu entschieden hat die einzelnen E-Mail-Typen als eigene Klassen abzubilden. Somit ist jeder E-Mail-Typ auch als eigener Java-Typ abgebildet. Am Beispiel der Klasse *CCItbCustUser* ist ebenso ersichtlich dass neben dem Abbilden eines E-Mail-Typs als eigene Java Klasse man ebenfalls eine eigene Subklassenhierarchie eingeführt hat um E-Mail-Typen, die 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 Klassenhierarchie. Eine Klassenhierarchie ist an sich starr und nicht flexibel und Änderungen an der Struktur können sich negativ in der Gesamtstruktur auswirken.

2.2.2 CCItbCustUser

Nachdem die Klassenhierarchien von *CCBasicEmail* diskutiert wurden wird im folgenden als Beispiel die Implementierung der Klasse *CCItbCustUser* angeführt. Diese Implementierung dient als Beispiel für die restlichen E-Mail-Typ Implementierungen, die nach dem selben Prinzip mit ähnlichen Umfang implementiert wurden. Im Punkt 2.2.1 wurde behauptet dass diese Ableitungen eingeführt wurden um E-Mail-Typen zu gruppieren. Man könnte aber auch davon ausgehen dass diese eigene Subklassenhierarchie eingeführt wurde um gemeinsame Funktionalitäten für die abgeleiteten E-Mail-Typen zu kapseln.

Folgender Source-Code illustriert dass die Implementierungen der einzelnen E-Mail-Typen hauptsächlich nur aus dem Aufbau der eigentlichen Nachricht besteht. Die Parameter für die Vorlage werden aus dem Resultat der spezifischen SQL-Abfrage extrahiert und in der Nachricht bzw. der verwendeten Vorlage verwendet. Die erstellte Nachricht wird dann als Resultat zurückgeliefert. Dies ist also der Grund für das Abbilden der einzelnen E-Mail-Typen als eigene Java Klassen. Dieser Ansatz produziert eine Unzahl an Klassen, die in einer starren Hierarchie gebunden sind und nur um die eigentliche Nachricht zu erstellen. Diese Klassen werden auch dazu verwendet um die E-Mail-Typen zu steuern. Jedoch ist hier das Erstellen der Nachricht zu stark an den E-Mail-Typ gekoppelt und es fehlt hier die Abstraktion.

Programm 2.1: Implementierung *CCItbCustUser*

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

Die einzelnen E-Mail-Typ Implementierungen implementieren lediglich die folgenden drei Methoden:

- *getMailType*
  Bereitstellen eines eindeutigen Schlüssels, der diesen E-Mail Typ identifiziert.
- *getMailBody*
  Erstellen der E-Mail Nachricht aus einer Vorlage, welche mit Parametern befüllt wird.
- *run()*
  Jeder E-Mail-Typ wird in einem eigenen Thread abgearbeitet. Hier wird entschiedenen welche Art von E-Mail Versand wird. (*CCBasiEmail* stellt mehrere Implementierungen zur Verfügung)

2.2.3 CCMailingDao

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

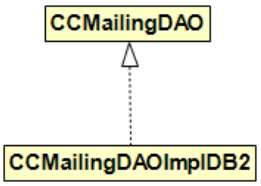

Abbildung 2.5: Klassenhierarchie von *CCMailingDao*

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

2.2.4 CCMailingDaoFactory

Zu kritisieren ist hier auch die Art und Weise wie eine Instanz von *CCMailingDao* erzeugt wird. Man nutzt hier den Factory-Pattern, jedoch wird hier statisch die zu verwendende Implementierung in *CCMailingDaoFactory* definiert, was das Austauschen der Implementierung unmöglich macht.

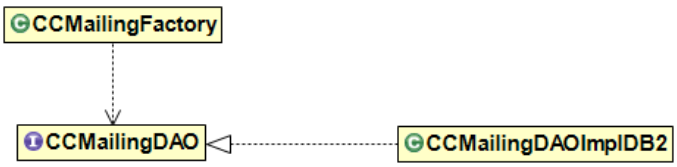

Abbildung 2.6: *CCMailingDaoFactory* für *CCMailingDao*

In dem Buch *Refactoring to patterns* [1, S. 72] wird als Nachteil einer Factory 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 Instanz über eine Getter-Methode oder die geschützte Member-Variable erreichen hätte man auf diese Factory verzichten können, da die Abstraktion bereits über die Basisklasse *CCBasiEmail* erreicht wurde.

Zusätzlich befindet sich die Schnittstelle zusammen mit ihrer Implementierung in ein und demselben Artefakt. Dies ist als ein halbherziger Versuch zu werten die Implementierung von *CCMailingDao* austauschbar zu machen.

2.3 Datenbank

Abschließend wird hier der Aufbau des Datenbankschemata betrachtet, welches die Kernkomponente des Systems darstellt aus der Sicht von *CCMail* darstellt. Es wurde bei diesem Schemata offensichtlich auf Fremdschlüssel verzichtet. In dem Buch *Refactoring Database* [2, 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 sein. Das Problem von nicht verwendeten Fremdschlüsseln wird in *Refactoring Databases* [2, 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 das 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 nciht zur Verfügung stand.

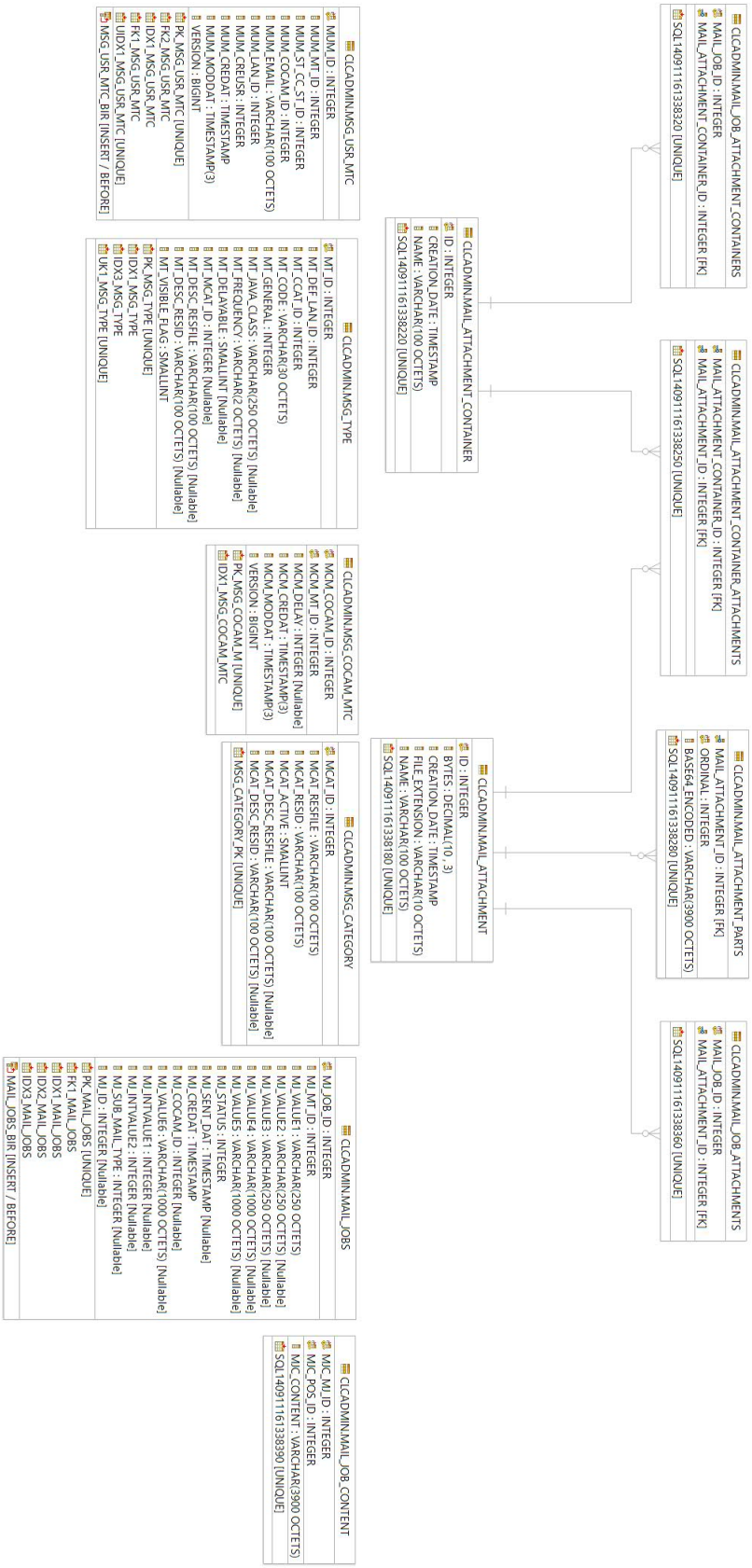

Abbildung 2.7: Datenbankschemata *CCMail*

Quellenverzeichnis

Literatur

[1] Joshua Kerievsky. *Refactoring to patterns.* München: Addison-Wesley Professional, 2005 (siehe S. 13).

[2] Scott W.Ambler und Pramond J.Sadalage. *Refactoring Databases.* München: Addison-Wesley Professional, 2006 (siehe S. 14).