



Fachhochschul-Bachelorstudiengang  
**SOFTWARE ENGINEERING**  
A-4232 Hagenberg, Austria

# **Vorlagenmanagement für *CleverMail***

Praktische  
Bachelorarbeit

zur Erlangung des akademischen Grades  
Bachelor of Science in Engineering

Eingereicht von

**Ing. Thomas Herzog**

Begutachtet von FH-Prof. DI Dr. Heinz Dobler

Hagenberg, August 2016

## **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.

Datum

Unterschrift

# Inhaltsverzeichnis

<b>Kurzfassung</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Unternehmen <i>curecomp Software Services GmbH</i> . . . . .	1
1.2 Vorlagenmanagement für <i>CleverMail</i> . . . . .	2
1.3 Rahmenbedingungen . . . . .	3
<b>2 Ziel des Projekts</b>	<b>4</b>
2.1 Funktionale Ziele . . . . .	5
2.1.1 Variablen für die Vorlagen . . . . .	5
2.1.2 Mehrsprachigkeit der Variablen . . . . .	5
2.1.3 Automatische Registrierung der Variablen . . . . .	5
2.1.4 Mehrsprachigkeit der Vorlagen . . . . .	6
2.1.5 Verwaltung der Vorlagen über eine Webseite . . . . .	6
2.2 Technische Ziele . . . . .	7
<b>3 Lösungskonzept</b>	<b>8</b>
3.1 Spezifikation des Vorlagenmanagements . . . . .	8
3.2 Spezifikation der Integration des Vorlagenmanagements . . .	19
3.2.1 Vorlagenmanagement im <i>CKEditor</i> . . . . .	19
3.2.2 Vorlagenmanagement in <i>CDI</i> . . . . .	20
3.2.3 Vorlagenmanagement in <i>JSF</i> . . . . .	20
<b>4 Realisierung</b>	<b>21</b>
4.1 Implementierung der Spezifikation . . . . .	23
4.1.1 Implementierungen für <i>CKEditor</i> . . . . .	23
4.1.2 Implementierungen für <i>CDI</i> . . . . .	29
4.1.3 Implementierungen für <i>JSF</i> . . . . .	35
4.2 Vorlagenmanagement-Beispielanwendung . . . . .	39
4.2.1 Verwendung über eine Webseite . . . . .	39
4.2.2 Verwendung in einer Geschäftslogik . . . . .	44

<b>5</b>	<b>Tests und Analyse</b>	<b>49</b>
5.1	Tests . . . . .	49
5.1.1	Tests der <i>CDI</i> -Integration . . . . .	50
5.1.2	Tests der <i>JSF</i> -Integration . . . . .	51
5.1.3	Tests des Vorlagenmanagements . . . . .	52
5.2	Analyse . . . . .	53
5.2.1	<i>CKEditor-Plugin</i> des Vorlagenmanagements . . . . .	53
5.2.2	<i>CDI</i> -Integration des Vorlagenmanagements . . . . .	54
5.2.3	<i>JSF</i> -Integration des Vorlagenmanagements . . . . .	54
<b>6</b>	<b>Zusammenfassung, weitere Aufgaben und Erfahrungen</b>	<b>55</b>
6.1	Zusammenfassung . . . . .	55
6.2	Weitere Aufgaben . . . . .	56
6.3	Erfahrungen . . . . .	56
	<b>Quellenverzeichnis</b>	<b>58</b>
	Literatur . . . . .	58
	Online-Quellen . . . . .	58

# Kurzfassung

Die vorliegende Bachelorarbeit behandelt das Vorlagenmanagement für die Anwendung *CleverMail*, die in der theoretischen Bachelorarbeit konzipiert wurde und eine Anwendung ist, die zum Versand von *E-Mails* verwendet wird. Mit dem Vorlagenmanagement können Vorlagen für *E-Mail*-Nachrichten zur Laufzeit und in mehreren Sprachen verwaltet werden.

Das Vorlagenmanagement verwendet mehrere Technologien und Sprachen wie *CDI*, *JSF* und *TypeScript*. Vor allem die Implementierung in Java 8 und die Möglichkeit der Verwendung der neuen sprachspezifischen Funktionalitäten wie *Lambda*-Ausdrücke, Methodenreferenzen und die *Stream-API* haben den Quelltext vereinfacht.

Die Integration des Vorlagenmanagements in eine *CDI*-Umgebung war einfach zu realisieren und hat gezeigt, dass ein Softwaremodul in eine *CDI*-Umgebung einfach integriert werden kann, sofern es die nötigen Voraussetzungen erfüllt. Die implementierte *CDI*-Erweiterung wird einfach zu erweitern sein und man könnte mehr Funktionalitäten, die in *CDI* zur Verfügung stehen, verwenden. Es könnten z.B. Erzeuger für Variablen registriert werden, die zur Laufzeit dynamisch Variablen erzeugen, anstatt die Variablen nur beim Start der *CDI*-Umgebung zu registrieren, welche dann über die Lebensdauer der *CDI*-Umgebung unveränderlich sind.

Während der Entwicklung des Vorlagenmanagements sind keine erwähnenswerten Probleme aufgetreten, alle Funktionalitäten und die Integration konnten einfach implementiert und getestet werden, wobei besonders die Einfachheit der Tests in einer *CDI*-Umgebung hervorgehoben werden muss, die mit der verwendeten Bibliothek *DeltaSpike* einfach aufgesetzt werden können und innerhalb einer Entwicklungsumgebung, ohne Anwendungsserver, lauffähig sind.

Die Implementierung des *CKEditor-Plugins* gestaltete sich einfach, da dieser *Editor* gut dokumentiert ist und es bereits Typinformationen für *TypeScript* gibt. Der *Editor TinyMCE*, für den anfangs das *Plugin* entwickelt werden

sollte, ist hingegen schlecht dokumentiert, daher wurde auf den *Editor CKE* gewechselt. Die Implementierung in *TypeScript* war die richtige Entscheidung, denn es hat die Entwicklung vereinfacht, und der Quelltext ist lesbarer als der Quelltext in *JavaScript*. Für die Zukunft wird *TypeScript* weitere sprachspezifische Möglichkeiten bieten, die den Quelltext noch mehr vereinfachen werden, obwohl eine Migration auf eine neuere Version von *TypeScript* zur Zeit nicht nötig ist.

# Abstract

This practical bachelor thesis is about the template management for the application *CleverMail*, which is an application designed for the theoretical bachelor thesis and which will be used for the sending of emails. With the template management email messages can be managed during runtime and for multiple languages.

The implemented template management uses several technologies and languages such as *CDI*, *JSF* and *TypeScript*. Especially the implementation in Java 8 and the possibility of the usage of the newly introduced language specific features such as Lambda expressions, Method references and the Stream API made the source more readable. The integration of the template management in a CDI environment was easy to accomplish and demonstrated, that a software module can be easily integrated in a CDI environment, if it meets the necessary requirements. The implemented CDI extension will be easy to extend and one could use more features provided by CDI. For example, producers for variables could be registered, which could dynamically produce variables during runtime, instead of registering the variables during the start of the CDI environment, which are then immutable over the lifetime of the CDI environment.

During the development of the template management no noteworthy problems occurred, all of the predefined features and the integration were easy to implement, whereby the simplicity of the tests within a CDI environment need to be emphasized, which can be set up easy and are executable within a development environment, without the need of an application server environment.

The implementation of the CKEditor plugin has proved to be easy, because the editor is well documented and there is already type information provided for TypeScript. The editor TinyMCE, which the plugin was supposed to be implemented in the first place, is poorly documented, which was the reason why we switched to the editor CKEditor. The implementation in TypeScript was a proper decision, because the source is more readable than the source

in JavaScript. TypeScript will provide more language specific features in the future, which will make the source even more understandable, although a migration to a newer version of TypeScript is not needed for now.



# Kapitel 1

## Einleitung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Spezifikation und der Implementierung eines Vorlagenmanagements für den in der theoretischen Bachelorarbeit konzipierten *Mail-Service* namens *CleverMail*. Das Vorlagenmanagement stellt einen essentiellen Teil von *CleverMail* dar, mit dem sich parametrisierte *E-Mail*-Vorlagen erstellen lassen. Das Vorlagenmanagement muss es den BenutzerInnen ermöglichen, einfach eigene parametrisierte *E-Mail*-Vorlagen zu erstellen, die in Anwendungen, die *CleverMail* nutzen, verwendet werden können, um benutzerdefinierte *E-Mail*-Nachrichten zu erstellen und zu versenden. Mit dem Vorlagenmanagement ist es nicht mehr erforderlich, die *E-Mail*-Vorlagen statisch zu definieren und die *E-Mail*-Vorlagen können von den BenutzerInnen nach ihren Wünschen angepasst werden.

Aufgrund des Umfangs von *CleverMail* wurde entschieden, sich vorerst auf das Vorlagenmanagement zu konzentrieren. Das Vorlagenmanagement wird für *CleverMail* entwickelt, könnte jedoch ohne weiteres auch in anderen Anwendungen verwendet werden, sofern diese Anwendungen die technischen Voraussetzungen erfüllen. Das Vorlagenmanagement wird als eigene Softwarekomponente entwickelt und wird keine Abhängigkeiten zu Ressourcen von *CleverMail* haben.

### 1.1 Unternehmen *curecomp Software Services GmbH*

Das Vorlagenmanagement wird in Zusammenarbeit mit dem Unternehmen *curecomp Software Services GmbH* erstellt. Das Unternehmen *curecomp* ist ein Dienstleister im Bereich des *Supplier Relationship Managements (SRM)* und betreibt eine eigene Softwarelösung namens *clevercure*. Die Softwarelösung *clevercure* besteht aus den folgenden Anwendungen:

- *CleverWeb* ist eine Webanwendung für den webbasierten Zugriff auf *clevercure*.
- *CleverInterface* ist eine Schnittstellenanwendung für den *XML*-basierten Datenimport und -export zwischen *clevercure* und den *ERP*-Systemen der Kunden und der Lieferanten.
- *CleverSupport* ist eine unternehmensinterne Webanwendung zur Unterstützung der Abwicklung von *Support*-Prozessen.
- *CleverDocument* ist ein Dokumentenmanagementsystem für die Verwaltung aller anfallender Dokumente innerhalb von *clevercure*.
- *CCMail* ist die bestehende *Mail*-Anwendung für den Versand der *E-Mails* innerhalb von *clevercure*, die durch *CleverMail* abgelöst werden soll.

Das Vorlagenmanagement wird von den Anwendungen innerhalb von *clevercure* verwendet werden, bevor *CleverMail* fertiggestellt wird, da es bereits Softwarekomponenten innerhalb der Anwendungen von *clevercure* gibt, die auf parametrierbare Vorlagen angewiesen sind.

## 1.2 Vorlagenmanagement für *CleverMail*

Mit dem Vorlagenmanagement können Vorlagen von den EntwicklerInnen und BenutzerInnen definiert und parametrierbar erstellt werden. Damit können Vorlagen dynamisch zur Laufzeit erstellt, modifiziert und gelöscht werden. Es sind keine statischen Vorlagen für die *E-Mail*-Nachrichten mehr nötig und alle damit verbunden Nachteile wie z.B.

- das neu Kompilieren und Einspielen bei Änderungen der Vorlagen,
- keine Möglichkeit für benutzerdefinierte Vorlagen oder
- keine Möglichkeit der Nutzung von dynamischen Parametern in den Vorlagen

sind nicht mehr vorhanden.

Das Vorlagenmanagement kann auch in einem anderen Kontext verwendet werden, wobei sich die vorliegende Bachelorarbeit ausschließlich mit der Verwendung des Vorlagenmanagements in *CleverMail* beschäftigt. Das Vorlagenmanagement wird als eigene Softwarekomponente implementiert und die vorliegende Bachelorarbeit zeigt auf, wie sich das Vorlagenmanagement in Anwendungen, im Kontext von *E-Mail*-Vorlagen, verwenden lässt.

### 1.3 Rahmenbedingungen

Das Vorlagenmanagement muss in Java 8 implementiert werden und muss die Plattform *Java Enterprise Edition 7 (JEE-7)* verwenden, wobei folgende Spezifikationen Anwendung finden müssen:

- *Java Persistence API 2.1 (JPA 2.1) (JSR 338)* ist die Spezifikation für die Persistenz in Java.
- *Context and Dependency Injection 1.1 (CDI 1.1) (JSR 346)* ist die Spezifikation für kontextabhängige Injektion innerhalb der Plattform *JEE-7*.
- *Java Server Faces 2.2 (JSF 2.2) (JSR 344)* ist die Spezifikation der *View-Technologie* in Java.

Damit wird das Vorlagenmanagement mit den aktuellsten Standards und Spezifikationen implementiert. Die Funktionalität des Vorlagenmanagements muss weitestgehend ohne die Verwendung von Bibliotheken von Drittanbietern implementiert werden. Das Vorlagenmanagement muss folgende Integration zur Verfügung stellen:

- Die Integration in *CDI*,
- die Integration in *JSF* und
- die Integration in *TypeScript*.

Als Entwicklungsumgebung wird *IntelliJ* [JetBrains 2016] verwendet, die eine bekannte Entwicklungsumgebung im *Java*-Umfeld ist und ein Produkt des Unternehmens *Jetbrains* mit Sitz in Tschechien ist.

Als Anwendungsserver wird *WildFly 10.0.0* [RedHat 2016], vormals *JBossAS* genannt, des Unternehmens *RedHat* verwendet, der ein zertifizierter *JEE-7*-Server ist und somit alle benötigten Spezifikationen unterstützt.

Es soll so weit wie möglich vermieden werden, Bibliotheken von Drittanbietern zu verwenden, außer sie sind für die Funktionalitäten des Vorlagenmanagements unerlässlich oder bieten einen essentiellen Vorteil.

## Kapitel 2

# Ziel des Projekts

Das Ziel des Projekts Vorlagenmanagement für *CleverMail* ist die Entwicklung der Softwarekomponente Vorlagenmanagement für die Verwendung in *CleverMail*, mit dem Vorlagen verwaltet werden können. Das Vorlagenmanagement stellt einen essentiellen Teil von *CleverMail* dar und wird auch von mehreren Anwendungen innerhalb von *clevercure* verwendet werden. Die verschiedenen Anwendungen, die das Vorlagenmanagement verwenden, sind ebenfalls in Java implementiert, werden aber in unterschiedlichen Laufzeitumgebungen betrieben werden wie z.B.:

- *IBM-Integration-Bus* (IIB) ist ein proprietäres Produkt des Unternehmens *IBM*, das für die *XML*-Konvertierungen und den *XML*-basierten Datenimport und -export verwendet wird.
- *WildFly* ist ein zertifizierter und frei verfügbarer *JEE-7* Anwendungsserver des Unternehmens *RedHat*.

Die verschiedenen Anwendungen von *clevercure* müssen mit möglichst wenig Aufwand in der Lage sein, Vorlagen zu verwenden und *E-Mail*-Nachrichten auf Basis dieser Vorlagen zu erstellen. Dabei müssen die Abhängigkeiten der Anwendungen zum Vorlagenmanagement so gering wie möglich gehalten werden, sowie nur vorgegebene Schnittstellen verwendet werden. Wird eine *E-Mail*-Nachricht von einer Anwendung auf Basis einer Vorlage erstellt, so müssen die aktuellen Werte der Variablen der Vorlage beim Zeitpunkt des Erstellens der *E-Mail*-Nachricht ermittelt und serialisiert werden, damit die *E-Mail*-Nachricht mit demselben Inhalt erneut versendet werden kann. Für die Anwendungen darf nicht erkennbar sein, wie die *E-Mail*-Nachrichten nach ihrer Erstellung weiter verwendet werden.

Zurzeit interagieren die Anwendungen direkt mit der Datenbank, anstatt von ihr abstrahiert zu sein und sind daher stark an die bestehende Anwendung *CCMail* gekoppelt bzw. an das Datenbankschema der Anwendung *CCMail*.

## 2.1 Funktionale Ziele

Für das Vorlagenmanagement wurden die folgenden funktionalen Ziele definiert, die umgesetzt werden müssen.

### 2.1.1 Variablen für die Vorlagen

Die Vorlagen werden für einen bestimmten *Mail*-Typ definiert, wobei eine Vorlage in einem bestimmten Kontext verwendet wird wie z.B.

- ein(e) BenutzerIn wurde erstellt,
- eine Bestellung wurde erstellt oder
- ein Dokument wurde hochgeladen.

Für die Vorlagen, die für einen bestimmten *Mail*-Typ erstellt werden, müssen Variablen zur Verfügung gestellt werden können wie z.B.:

- Die Variable *CURRENT\_USER* ist der Benutzer, der die *E-Mail*-Nachricht erstellt halt.
- Die Variable *ORDER\_NUMBER* ist die Nummer der erstellten Bestellung.

Die EntwicklerInnen müssen für einen bestimmten *Mail*-Typ in der Lage sein, einfach Variablen zu definieren, die von den BenutzerInnen, beim Erstellen einer Vorlage für den korrespondierenden *Mail*-Typ, frei verwendet werden können. Die Variablen müssen auch global definiert werden und prinzipiell in allen Vorlagen verwendbar sein. Die EntwicklerInnen müssen in der Lage sein, die Menge der zur Verfügung stehenden Variablen zur Laufzeit, aufgrund von bestimmten Zuständen, verändern zu können. Die Menge der Variablen könnte z.B. von Berechtigungen der BenutzerInnen abhängig sein.

### 2.1.2 Mehrsprachigkeit der Variablen

Die zur Verfügung stehenden Variablen werden durch die EntwicklerInnen statisch definiert und müssen jeweils einen Bezeichner, eine Bezeichnung und eine Beschreibung zur Verfügung stellen. Die Bezeichnung und die Beschreibung einer Variable müssen mehrsprachig zur Verfügung stehen, wobei als Standardsprache Englisch zu verwenden ist. Die Mehrsprachigkeit soll über *Java-Properties*-Dateien abgebildet werden, wobei als Zeichenkodierung *UTF8* zu verwenden ist, obwohl *Java-Properties*-Dateien laut Spezifikation die Zeichenkodierung *ISO 8859-1* verwenden müssen.

### 2.1.3 Automatische Registrierung der Variablen

Innerhalb einer *CDI*-Umgebung sollen die definierten Variablen beim Start der *CDI*-Umgebung automatisch gefunden und registriert werden. Die au-

tomatische Registrierung der Variablen muss mit einer *CDI*-Erweiterung realisiert werden, die beim Start der *CDI*-Umgebung die Variablen findet, registriert und über die Anwendungslebensdauer persistent hält. Mit einer automatischen Registrierung der Variablen wird erreicht, dass neu definierte Variablen automatisch gefunden und registriert werden und somit nicht manuell registriert werden müssen. Ein manuelles Registrieren der Variablen birgt das Risiko in sich, dass Variablen vergessen werden könnten.

#### 2.1.4 Mehrsprachigkeit der Vorlagen

Die Vorlagen müssen in mehreren Sprachen erstellt und verwaltet werden können, wobei eine Sprache als Standardsprache zu definieren ist, und es für diese Sprache immer einen Eintrag geben muss. Auf die Standardsprache wird zurückgegriffen, wenn es für eine angeforderte Sprache keinen Eintrag gibt. Somit ist gewährleistet, dass für jede angeforderte Sprache immer eine Vorlage zur Verfügung steht. Es ist nicht erforderlich, dass die Menge und Position der Variablen in einer Vorlage über alle definierten Sprachen gleich sind. Es dürfen in einer Vorlage, die in mehreren Sprachen definiert wurde, eine unterschiedliche Anzahl von Variablen, unterschiedliche Variablen und unterschiedliche Positionen der Variablen definiert sein.

#### 2.1.5 Verwaltung der Vorlagen über eine Webseite

Die Vorlagen müssen über eine Webseite verwaltet werden können. Die Webseite muss mit der *View*-Technologie *JSF* implementiert werden. Über einen *FacesConverter* soll die Vorlage von ihrer *HTML*-Repräsentation in die Repräsentation der verwendeten *Template-Engine* konvertiert werden und vice versa. Der Quelltext 2.1 zeigt ein *HTML-Markup* einer Vorlage, wie es in der Webseite bzw. innerhalb des *Editors CKEditor* verwendet wird. Die Variablen werden als *HTML-Tags* repräsentiert, aus denen die Variablen wieder ermittelt werden können.

**Quelltext 2.1:** Das *HTML-Markup* einer Vorlage

```
<p>Das ist eine Variable:</p>
<p>
  <span class="variable"
    title="Die Beschreibung der Variable"
    data-variable="VAR_ID">
    Die Bezeichnung der Variable
  </span>
</p>
```

Der Quelltext 2.2 zeigt das konvertierte *HTML-Markup* des Quelltexts 2.1 als *Freemarker-Vorlage* .

**Quelltext 2.2:** Das konvertierte *HTML-Markup* als *Freemarker-Vorlage*

```
<p>Das ist eine Variable:</p>
<p>
    ${module.core.VariableHolder["VAR_ID"]!("Nicht verfügbar")}
</p>
```

In der Webseite muss der *JavaScript*-basierte *CKEditor* [CKSource 2016] verwendet werden, weil für den *CKEditor* durch *PrimeFaces-Extensions* [PrimeFaces-Extensions 2016] eine *JSF*-Integration, in Form einer vollständigen *JSF*-Komponente, zur Verfügung gestellt wird. Es muss auch deshalb der *Editor CKEditor* verwendet werden, weil eine Integration in den Lebenszyklus von *JSF* notwendig ist, damit z.B. auf *AJAX*-Anfragen reagiert werden kann, wie es in *JSF* üblich ist.

## 2.2 Technische Ziele

Es wurden die im Folgenden aufgelisteten technischen Ziele definiert:

- Die Entwicklung in Java 8,
- die Entwicklung mit der Plattform *JEE-7*,
- die Integration in eine *CDI*-Umgebung,
- die Integration in *JSF*,
- die Integration in *TypeScript* und
- die Entwicklung als eigene Softwarekomponente.

Das Vorlagenmanagement muss Schnittstellen definieren, welche die Funktionalität des Vorlagenmanagements nach außen offenlegen, ohne dass die Anwendungen in Berührung mit den konkreten Implementierungen kommen.

## Kapitel 3

# Lösungskonzept

Dieses Kapitel beschäftigt sich mit der Spezifikation des Vorlagenmanagements. Bei der Spezifikation handelt es sich um die Schnittstellen und die abstrakten Klassen, welche die Struktur des Vorlagenmanagements definieren und die gemeinsame Logik vorgeben. Diese Schnittstellen und abstrakten Klassen erlauben es, Implementierungen für verschiedene *Template-Engines* zur Verfügung zu stellen wie z.B. für

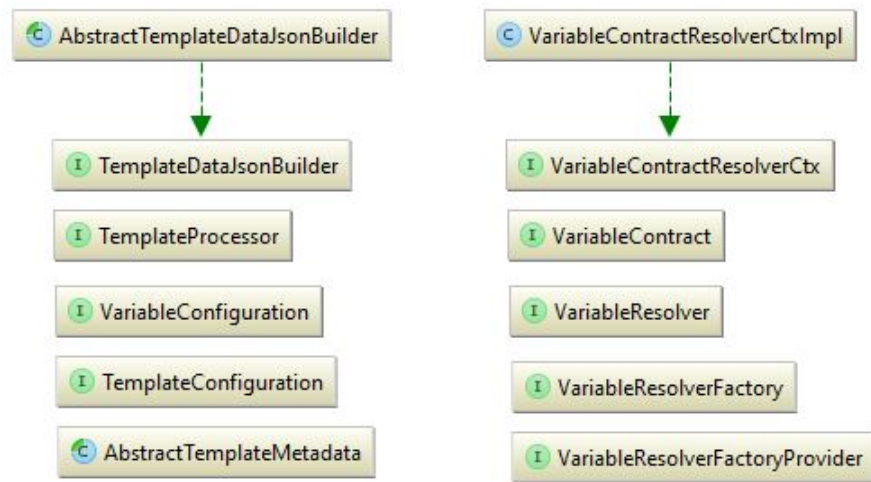
- die *Template-Engine Freemakrer*,
- die *Template-Engine Velocity* oder
- die *Template-Engine Thymeleaf*.

Mit der Möglichkeit, verschiedene *Template-Engines* unterstützen zu können, soll das Vorlagenmanagement flexibel gehalten werden. Bei einem Wechsel zu einer anderen *Template-Engine*, müssen nur die Ausdrücke in einer Vorlage in die *Template-Engine*-spezifischen Ausdrücke konvertiert werden, was sich einfach realisieren lässt, da die Ausdrücke einer Vorlage immer gefunden werden müssen.

### 3.1 Spezifikation des Vorlagenmanagements

Dieser Abschnitt behandelt die Spezifikation des Vorlagenmanagements. Auf Basis dieser Spezifikation wird das Vorlagenmanagement und die Integration in die verschiedenen Umgebungen und Technologien implementiert. Diese Spezifikation ist frei von Abhängigkeiten auf konkrete Implementierungen jeglicher Art. Sie hat nur Abhängigkeiten auf andere Spezifikationen wie z.B. die *JEE-7*-Spezifikation.





**Abbildung 3.1:** Klassenhierarchie des Vorlagenmanagements

Die Abbildung 3.1 zeigt die definierten Schnittstellen und Klassen des Vorlagenmanagements. Die abstrakten Klassen implementieren die gemeinsam nutzbare Logik, die von allen konkreten Implementierungen des Vorlagenmanagements für jede *Template-Engine* genutzt werden können. Die Schnittstellen und Klassen aus Abbildung 3.1 spezifizieren die Aspekte des Vorlagenmanagements, wie

1. das Variablenmanagement innerhalb des Vorlagenmanagements,
2. die Handhabung von Variablen in einer Vorlage,
3. die Abbildung der Metadaten einer Vorlage und
4. das Erstellen des *JSON*-Datenobjekts, welches die serialisierten Daten der Variablen einer Vorlage, sowie die Metadaten der Vorlage, enthält.

### **Schnittstelle *VariableContract***

Die Schnittstelle *VariableContract* aus dem Quelltext 3.1 spezifiziert eine Variable, die in einer Vorlage verwendet werden kann. Objekte dieses Datentyps werden beim Anwendungsstart registriert und können grundsätzlich in allen Vorlagen verwendet werden. Eine Variable ist einem Modul zugeordnet, wobei die Variable bezüglich ihres Namens innerhalb des Moduls eindeutig sein muss. Das Modul wird über eine Zeichenkette definiert. Die Mehrsprachigkeit einer Variable wird über einen Aufzählungstyp realisiert, wobei jede Variable jeweils einen Schlüssel für die Bezeichnung und die Beschreibung bereit stellen muss.

Da es sich bei den Variablen um statische Daten handelt, also die Variablen schon beim Kompilieren bekannt sind, ist angedacht, dass die Variablen als

eigener Aufzählungstyp implementiert werden, der die Schnittstelle *VariableContract* implementiert. Durch die Abbildung der Variablen über einen Aufzählungstyp können mehrere Variablen in einer Klasse definiert werden, wobei jede einzelne Aufzählung des Aufzählungstyps ein Objekt des Datentyps *VariableContract* darstellt. Alle Variablen, die mit einem Aufzählungstyp definiert werden, sollten demselben Modul zugeordnet sein, obwohl dies nicht zwingend erforderlich ist. Die Variablen, die mit einem Aufzählungstyp definiert wurden, werden innerhalb des Vorlagenmanagements trotzdem als einzelne Objekte des Datentyps *VariableContract* betrachtet. Die Tatsache, dass die Variablen mit einem Aufzählungstyp abgebildet wurden, ist für das Vorlagenmanagement nur beim Registrieren der Variablen von Belang und nicht bei deren weiterer Verwendung.

Eine Variable ist über ihren Bezeichner global eindeutig identifizierbar, wobei sich der Bezeichner aus dem Modulnamen und dem Variablennamen zusammensetzt (Bsp. *module.core.VAR\_1*). Der Bezeichner sowie der Modulname müssen sich an die Namenskonvention eines *Java*-Paketnamens halten. Da der Variablenname immer auf diese Weise zusammengesetzt werden muss, wurde die Methode *getId* als *Default*-Methode implementiert, was seit Java 8 möglich ist. Ein(e) EntwicklerIn muss diese Methode nicht mehr implementieren, obwohl es immer noch möglich ist diese Methode zu überschreiben. Auch die Methode *toInfoString* wurde als *Default*-Methode implementiert, da auch diese Methode nicht von den EntwicklerInnen implementiert werden sollte, da ihre Funktionalität sich nicht ändern sollte.

[Raoul-Gabriel u. a. 2014, S. 213] erklären *Default*-Methoden wie folgt.

*Default methods are a new feature added in Java 8 to help evolve APIs in a compatible way. An interface can now contain method signatures for which an implementing class doesn't provide an implementation. So who implements them? The missing method bodies are given as part of the interface (hence default implementations) rather than in the implementing class.*

Die Schnittstelle *VariableContract* definiert *Default*-Methoden nicht wegen einer Erweiterung der Schnittstelle, sondern wegen dem gleichen Verhalten der Methoden für alle Implementierungen.

**Quelltext 3.1:** Die Schnittstelle *VariableContract*

```
1 public interface VariableContract extends Serializable {
2
3     String getName();
4
5     String getModule();
6
7     Enum<?> getInfoKey();
8
9     Enum<?> getLabelKey();
10
11     default String getId() {
12         return getModule() + "." + getName();
13     }
14
15     default String toInfoString() {
16         final String ls = System.lineSeparator();
17         final StringBuilder sb = new StringBuilder();
18         sb.append("contract  : ").append(this.getClass().getName())
19           .append(ls)
20           .append("id        : ").append(getId())
21           .append(ls)
22           .append("name      : ").append(getName())
23           .append(ls)
24           .append("label-key : ").append((getLabelKey() != null)
25                               ? getLabelKey().name()
26                               : "not available")
27           .append(ls)
28           .append("info-key  : ").append((getInfoKey() != null)
29                               ? getInfoKey().name()
30                               : "not available")
31           .append(ls)
32           .toString();
33     }
34
35 }
```

**Schnittstelle *VariableResolver***

Die Schnittstelle *VariableResolver* aus dem Quelltext 3.2 spezifiziert, wie der aktuelle Wert der Variablen ermittelt wird.

Beim Erstellen einer *E-Mail*-Nachricht auf Basis einer Vorlage müssen die aktuellen Werte der Variablen der Vorlage ermittelt werden. Da der aktuelle Wert einer Variable kontextabhängig ist, wird beim Ermitteln des aktuellen Werts einer Variable, ein Kontextobjekt bereitgestellt, über welches kontextabhängige Daten bereitgestellt werden. Durch dieses Kontextobjekt, kann eine Variable in mehreren Kontexten verwendet werden und auch der

aktuelle Wert einer Variable kontextabhängig ermittelt werden.

**Quelltext 3.2:** Die Schnittstelle *VariableResolver*

```
1 @FunctionalInterface
2 public interface VariableResolver {
3
4     String resolve(VariableContract variable,
5                   VariableContractResolverCtx ctx);
6
7 }
```

Die Schnittstelle *VariableResolver* ist eine funktionale Schnittstelle, also eine Schnittstelle, die nur eine abstrakte Methode definiert, die implementiert werden muss. Eine Implementierung einer funktionalen Schnittstelle kann auch über einen *Lambda*-Ausdruck oder eine Methodenreferenz bereitgestellt werden, wodurch die Notwendigkeit einer anonymen Implementierung oder der Implementierung einer Klasse für diese Schnittstelle entfällt. Die Verwendung von *Lambda*-Ausdrücken und Methodenreferenzen macht den Quelltext lesbarer, obwohl angemerkt sei, dass dieser Ansatz sich negativ auf das Laufzeitverhalten auswirkt, was in der Art und Weise der Ausführung eines *Lambda*-Ausdrucks oder einer Methodenreferenz begründet ist. Die negativen Auswirkungen auf das Laufzeitverhalten können, in Bezug auf das Vorlagenmanagement, vernachlässigt werden. [Raoul-Gabriel u. a. 2014, S. 50] beschreiben den Nutzen von funktionalen Schnittstellen wie folgt.

*Functional interfaces are useful because the signature of the abstract method can describe the signature of a lambda expression. The signature of the abstract method of a functional interface is called a function descriptor.*

**Schnittstelle *VariableResolverFactory***

Die Schnittstelle *VariableResolverFactory* aus dem Quelltext 3.3 spezifiziert wie Objekte des Datentyps *VariableResolver* produziert werden. Objekte des Datentyps *VariableResolverFactory* können Objekte des Datentyps *VariableResolver* für jede Implementierung der Schnittstelle *VariableContract* produzieren. Es wird aber empfohlen, dass es je eine Implementierung der Schnittstelle *VariableResolverFactory* je Modul gibt.

**Quelltext 3.3:** Die Schnittstelle *VariableResolverFactory*

```
1 @FunctionalInterface
2 public interface VariableResolverFactory extends Serializable {
3
4     VariableResolver getVariableResolver(VariableContract contract,
5                                         VariableContractResolverCtx ctx);
6
7 }
```

Die Schnittstelle *VariableResolver* ist eine funktionale Schnittstelle, damit Implementierungen über *Lambda*-Ausdrücke oder Methodenreferenzen bereitgestellt werden können.

**Schnittstelle *VariableResolverFactoryProvider***

Die Schnittstelle *VariableResolverFactoryProvider* aus dem Quelltext 3.4 spezifiziert, wie Objekte des Datentyps *VariableResolverFactory* produziert werden. Ein Objekt des Datentyps *VariableResolverFactoryProvider* kann Objekte des Datentyps *VariableResolverFactory* für die Schnittstelle *VariableContract*, einer Ableitung von dieser Schnittstelle oder einer konkreten Implementierung dieser Schnittstelle zur Verfügung stellen. Die Schnittstelle *VariableResolverFactoryProvider* wurde spezifiziert, damit in einer *CDI*-Umgebung über ein Objekt dieses Datentyps die Objekte des Datentyps *VariableResolverFactory* produziert werden können, die von der *CDI*-Umgebung zur Verfügung gestellt werden.

**Quelltext 3.4:** Die Schnittstelle *VariableResolverFactoryProvider*

```
1 @FunctionalInterface
2 public interface VariableResolverFactoryProvider extends Serializable {
3
4     VariableResolverFactory getVariableResolverFactory
5         (Class<? extends VariableContract> contractType);
6
7 }
```

Die Schnittstelle *VariableResolverFactoryProvider* ist ebenfalls eine funktionale Schnittstelle, damit Implementierungen über *Lambda*-Ausdrücke oder Methodenreferenzen bereitgestellt werden können.

### Schnittstelle *VariableContractResolverCtx*

Die Schnittstelle *VariableContractResolverCtx* aus dem Quelltext 3.5 spezifiziert den Kontext, der beim Ermitteln des aktuellen Werts einer Variable zur Verfügung gestellt wird. Dieser Kontext stellt alle Daten bereit, die beim Ermitteln des aktuellen Werts einer Variable benötigt werden. Es wird auch ermöglicht, dass Benutzerdaten im Kontext definiert werden können, die bei beim Ermitteln des aktuellen Werts einer Variable verwendet werden können. Es wurde bewusst vermieden, dass beim Ermitteln eines aktuellen Werts einer Variable bekannt ist, in welcher Vorlage die Variable verwendet wird. Dadurch bleibt die Handhabung der Variablen einer Vorlage entkoppelt von der Vorlage selbst. Dadurch wäre es z.B. auch möglich die Variablen außerhalb des Vorlagenmanagements zu verwenden.

**Quelltext 3.5:** Die Schnittstelle *VariableContractResolverCtx*

```
1 public interface VariableContractResolverCtx {  
2  
3     Locale getLocale();  
4  
5     ZoneId getZoneId();  
6  
7     TimeZone getTimeZone();  
8  
9     <T> T getUserData(Object key, Class<T> clazz);  
10  
11 }
```

### Schnittstelle *TemplateProcessor*

Die Schnittstelle *TemplateProcessor* aus dem Quelltext 3.7 spezifiziert, wie die Variablen in einer Vorlage behandelt werden. Objekte dieses Datentyps können Variablen in einer Vorlage für eine bestimmte *Template-Engine* finden und konvertieren. Ein Objekt des Datentyps *TemplateProcessor* muss in der Lage sein, ungültige Variablen innerhalb einer Vorlage zu finden, wobei eine ungültige Variable, eine Variable ist, die nicht registriert ist. Eine Implementierung der Schnittstelle *TemplateProcessor* ist eine Implementierung für eine bestimmte *Template-Engine*, da die Variablen in Form von Ausdrücken spezifisch für die verwendete *Template-Engine* in der Vorlage enthalten sind.

Der Quelltext 3.6 zeigt die beiden Methoden der Schnittstelle *TemplateProcessor*, welche die Variablen in einer Vorlage konvertieren können.

**Quelltext 3.6:** Die Methoden für die Konvertierung

```
String replaceExpressions(  
    String template,  
    Function<VariableContract, String> converter);  
  
String replaceCustom(String template,  
    Pattern itemPattern,  
    Function<String, String> converter);
```

Die Methoden aus dem Quelltext 3.6 definieren als Formalparameter für den benötigte Konverter die funktionale Schnittstelle *Function*, welche von Java 8 bereitgestellt wird. Dadurch ist das Spezifizieren einer eigenen Schnittstelle für die Konvertierung nicht mehr nötig. Der Konverter kann über einen *Lambda*-Ausdruck oder eine Methodenreferenz bereitgestellt werden. Dadurch ist die Konvertierung der Variablen einer Vorlage abstrahiert von der Implementierung der Schnittstelle *TemplateProcessor*, wodurch die Variablen durch eine beliebige Repräsentation ersetzt werden können.

**Quelltext 3.7:** Die Schnittstelle *TemplateProcessor*

```
1 public interface TemplateProcessor {  
2  
3     String replaceExpressions(  
4         String template,  
5         Function<VariableContract, String> converter);  
6  
7     String replaceCustom(String template,  
8         Pattern itemPattern,  
9         Function<String, String> converter);  
10  
11     Set<VariableContract> resolveExpressions(String template);  
12  
13     Set<String> resolveInvalidExpressions(String template);  
14  
15     String variableToExpression(VariableContract contract);  
16  
17     VariableContract expressionToVariable(String expression);  
18  
19 }
```

**Schnittstelle *TemplateDataJsonBuilder***

Die Schnittstelle *TemplateDataJsonBuilder* aus dem Quelltext 3.8 spezifiziert die Signatur eines *Builders*, der das Datenobjekt erstellt, welches die

Daten für das Ausprägen einer Vorlage enthält. Das erstellte Datenobjekt enthält die folgenden Daten:

- Die Sprache, in der die Vorlage erstellt wurde,
- die Zone für die Konvertierung von Datums- und Zeitwerten,
- die Version der Vorlage und
- die Metadaten der Vorlage wie z.B die Anzahl der enthaltenen Variablen.

Das Datenobjekt kann in den folgenden Repräsentationen vom *Builder* bereitgestellt werden:

- als *Java*-Objekt,
- als *JSON*-Zeichenkette oder
- als Objekt der Klasse *java.util.Map*.

Anstelle der Serialisierung der Daten könnte die Vorlage auch ausgeprägt und persistent gehalten werden, wodurch aber die Menge an persistent gehaltenen Daten stark ansteigen würde. Mit dem Datenobjekt werden nur die benötigten Daten persistent gehalten, wodurch die Menge an persistent gehaltenen Daten so klein wie möglich gehalten wird. Mit dem Datenobjekt kann die korrespondierende Vorlage zu jedem Zeitpunkt mit demselben Resultat wiederhergestellt werden.

Es wurde das Entwurfsmuster *Builder* [Gamma u. a. 1994, S. 97] verwendet, da sich die Konfiguration des *Builders* mit einer *Fluent*-Schnittstelle [Fowler 2005], wie bei einem *Builder* üblich, sehr gut abbilden lässt. Die Schnittstelle *TemplateDataJsonBuilder* spezifiziert folgende Terminalmethoden.

- *TemplateRequestJson toJsonModel()* liefert das Datenobjekt in Form eines *Java*-Objekts.
- *String toJsonString()* liefert das Datenobjekt als *JSON*-Zeichenkette.
- *Map<String, Object> toJsonMap()* liefert das Datenobjekt in Form eines Objekts der Klasse *java.util.Map*.



**Quelltext 3.8:** Die Schnittstelle *TemplateDataJsonBuilder*

```
1 public interface TemplateDataJsonBuilder<
2     I,
3     M extends AbstractTemplateMetadata<I>,
4     B extends TemplateDataJsonBuilder> extends Serializable {
5
6     B withWeakMode();
7
8     B withLocalization(Locale locale,
9                        ZoneId zoneId);
10
11    B withUserData(Map<Object, Object> userData);
12
13    B withStrictMode();
14
15    B withVariableResolverFactoryProvider
16        (VariableResolverFactoryProvider factory);
17
18    B withVariableResolverFactory(VariableResolverFactory factory);
19
20    B withTemplate(M metadata);
21
22    B addVariable(VariableContract contract, Object value);
23
24    B addVariableResolver(VariableContract contract,
25                        VariableResolver resolver);
26
27    TemplateRequestJson toJsonModel();
28
29    String toJsonString();
30
31    Map<String, Object> toJsonMap();
32
33 }
```

[Gamma u. a. 1994, S. 100] beschreiben die Implementierung eines *Builders* wie folgt:

*Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default. A ConcreteBuilder class overrides operations for components it's interested in creating.*

[Gamma u. a. 1994, S. 96 - 106] beschreiben ausführlich das Entwurfsmuster *Builder*, jedoch ohne die Verwendung einer *Fluent*-Schnittstelle, die heutzutage über den *Builder* gelegt wird, um dessen Anwendung über Punktnotation angenehmer zu machen. Mit einer *Fluent*-Schnittstelle wird die

Möglichkeit geboten, einzelne Komponenten des zu bauenden Objektes zu setzen. Der Quelltext 3.9 zeigt, wie der *Builder* des Datentyps *TemplateDataJsonBuilder* mit einer *Fluent*-Schnittstelle verwendet wird.

**Quelltext 3.9:** Beispiel der Anwendung des *Builders*

```
1 TemplateDataJsonBuilder builder;  
2 TemplateRequestJson    model;  
3  
4 builder = new TemplateDataJsonBuilderImpl();  
5 model   = builder.withStrictMode()  
6             .withLocalization(localeObj, zoneIdObj)  
7             .withTemplate(templateMetadataObj)  
8             .withUserData(userDataMap)  
9             .withVariableResolverFactoryProvider(factoryProviderObj)  
10            .toJsonModel();
```

### Abstrakte Klasse *AbstractTemplateMetadata*

Die abstrakte Klasse *AbstractTemplateMetadata* implementiert die Logik, die von allen konkreten Ableitungen dieser abstrakten Klasse für die verschiedenen *Template-Engines* genutzt werden kann. Metadaten wie

- die Anzahl der gültigen Variablen in der Vorlage,
- die Anzahl der nicht registrierten Variablen in der Vorlage,
- die Zeichenanzahl der Vorlage,
- der eindeutige Bezeichner der Vorlage,
- die Version der Vorlage und
- die Vorlage selbst,

werden in dieser Klasse abgebildet.

Diese Metadaten sind unabhängig von der verwendeten *Template-Engine* und eine Implementierung für eine spezifische *Template-Engine* kann zusätzliche Metadaten definieren. Die Metadaten werden einmalig ermittelt und sind über die Lebenszeit des Objekts des Datentyps *AbstractTemplateMetadata* unveränderbar. Wird die Vorlage geändert so muss entweder ein neues Objekt erstellt werden oder das Objekt muss neu initialisiert werden.

### Abstrakte Klasse *AbstractTemplateDataJsonBuilder*

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder* implementiert die gemeinsam nutzbare Logik, die von allen konkreten Ableitungen für die verschiedenen *Template-Engines* verwendet werden kann. Sie stellt Hilfsmethoden bereit, die Variablen innerhalb der Vorlage finden, validieren und den

aktuellen Wert von Variablen ermitteln können. Das resultierende Datenobjekt des *Builders* ist spezifiziert, jedoch nicht die Abbildung der ermittelten Werte für die enthaltenen Variablen. Diese Daten sind spezifisch für die verwendete *Template-Engine*.

## 3.2 Spezifikation der Integration des Vorlagenmanagements

Die im Abschnitt 3.1 vorgestellte Spezifikation des Vorlagenmanagements, spezifiziert die Kernfunktionalität des Vorlagenmanagements, das in der Lage ist, die Vorlagen sowie deren enthaltene Variablen zu behandeln. Das Vorlagenmanagement benötigt auch die Integration in verschiedene Umgebungen und Sprachen, um die benötigten Funktionalitäten wie

- die Verwaltung der Variablen im *JavaScript*-basierten *CKEditor*,
- die automatische Registrierung der Variablen in einer *CDI*-Umgebung,
- die Verwaltung der Vorlagen über eine Webseite und
- die Persistenz der Vorlagen

realisieren zu können.

Folgende Abschnitte behandeln die Spezifikation der Integration wie in Abschnitt 2.2 vorgegeben.

### 3.2.1 Vorlagenmanagement im *CKEditor*

Wie in Abschnitt 2.1.5 vorgegeben, muss der *JavaScript*-basierte *Editor CKEditor* verwendet werden, mit dem *HTML* basierte Vorlagen über eine Webseite bearbeitet werden können. Der *CKEditor* muss angepasst werden, damit die definierten Variablen in einer Vorlage verwendet werden können.

Dazu wird ein *CKEditor-Plugin* in *TypeScript* entwickelt, welches es erlaubt, die definierten Variablen innerhalb des *CKEditors* und dessen enthaltener Vorlage zu verwalten. Es wird die Skriptsprache *TypeScript* verwendet, da es mit dieser Skriptsprache möglich ist, typischer zu entwickeln, was in *JavaScript* nicht möglich ist. Ebenfalls kann *TypeScript* in mehrere *ECMA*-Standards übersetzt werden.

Innerhalb des *CKEditor-Plugins* werden Variablen verwendet, wobei das Variablenmanagement in einer eigenen Quelltextdatei implementiert wird, da das Variablenmanagement unabhängig vom *CKEditor-Plugin* ist und daher auch anderweitig verwendet werden kann. Damit wird das Variablenmanagement vom *CKEditor-Plugin* entkoppelt.

### 3.2.2 Vorlagenmanagement in *CDI*

Das Vorlagenmanagement wird in einem *JEE-7*-Anwendungsserver verwendet, der eine *CDI*-Umgebung bereitstellt. Im *CDI*-Standard sind portable Erweiterungen spezifiziert, die es erlauben, dass sich Softwarekomponenten in einer *CDI*-Umgebung integrieren können. Es wird eine *CDI*-Erweiterung implementiert, die beim Start der *CDI*-Erweiterung, die definierten Variablen automatisch registriert und über den Lebenszyklus der Anwendung persistent hält. Es werden Objekte des Vorlagenmanagements wie z.B.:

- Objekte des Datentyps *VariableResolver*,
- Objekte des Datentyps *VariableResolverFactory* oder
- Objekte des Datentyps *TemplateDataJsonBuilder*

kontextabhängig zur Verfügung gestellt.

Durch die Verwaltung der Objekte des Datentyps *VariableResolver* von einer *CDI*-Umgebung, können sich Implementierungen der Schnittstelle *VariableResolver* kontextabhängige Ressourcen injizieren lassen. Damit das Variablenmanagement auf diese Objekte zugreifen kann, wurde die Schnittstelle *VariableResolverFactoryProvider* spezifiziert, welche die Verbindung des Variablenmanagements zu einer *CDI*-Umgebung herstellt und kontextabhängige Objekte des Datentyps *VariableResolverFactory* bereitstellen kann.

### 3.2.3 Vorlagenmanagement in *JSF*

Für die Verwaltung der Vorlagen wird eine *JSF*-Webseite implementiert. Über diese Webseite können Vorlagen erstellt, modifiziert und gelöscht werden.

Für die Verwaltung der Vorlagen wird die von *PrimeFaces-Extension* bereitgestellte *JSF*-Komponente für den Editor *CKEditor* verwendet. Diese Komponente integriert den *JavaScript*-basierten *CKEditor* in den *JSF*-Lebenszyklus.

Um die Vorlage in die korrespondierende *Template-Engine*-spezifische Repräsentation zu überführen, wird ein *FacesConverter* implementiert, der die Konvertierung der Vorlage von seiner Repräsentation in *HTML* in die *Template-Engine* spezifische Repräsentation und vice versa ermöglicht.

## Kapitel 4

# Realisierung

Dieses Kapitel befasst sich mit der Implementierung der Spezifikation des Vorlagenmanagements, die im Kapitel 3 vorgestellt wurde. Die Implementierung wurde in Java 8 mit dem *Buildtool Maven* realisiert, wobei die Implementierungen in der Projektstruktur organisiert wurden, die in Abbildung 4.1 dargestellt ist.

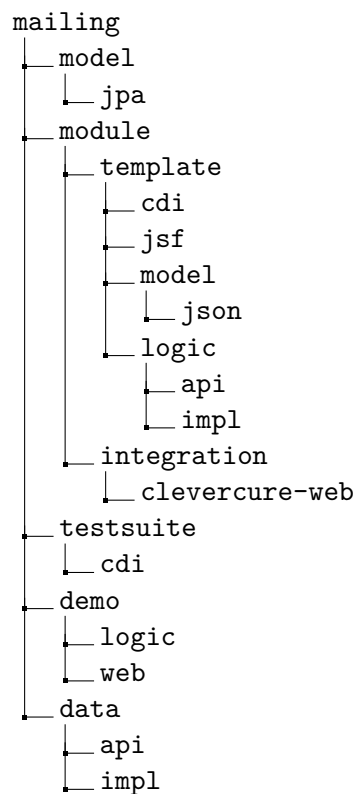


Abbildung 4.1: Die Verzeichnisstruktur der *Maven*-Projekte

Das *Maven*-Wurzelprojekt *mailing* organisiert alle benötigten Abhängigkeiten, sowie die auf alle Unterprojekte anwendbare *Build*-Konfigurationen. Die übergeordneten Projekte sind vom Typ *pom*, was bedeutet, dass aus diesen Projekten keine Artefakte erstellt werden können und die übergeordneten Projekte die tiefer liegenden Projekte bündeln. Die gesamte Organisation der Abhängigkeiten findet im Wurzelprojekt *mailing* statt. Diese Projektstruktur wurde gewählt, da in diesem Projekt auch die Implementierungen der anderen Softwarekomponenten von *CleverMail* organisiert werden.

Die konkreten Artefakte wurden jeweils in ein Artefakt *\*-api* und *\*-impl* aufgeteilt, somit sind die Schnittstellen vollständig getrennt von deren Implementierungen.

Folgende Auflistung beschreibt alle konkreten Artefakte (*Java-Archive*), die aus dem Wurzelprojekt *mailing* erstellt werden können.

- ***mailing-model-jpa*** enthält die Klassen mit den *JPA*-Entitäten, welche die Datenbank in *Java* abbilden.
- ***mailing-module-template-cdi*** enthält die Implementierung der *CDI*-Erweiterung für die Integration in eine *CDI*-Umgebung.
- ***mailing-module-template-jsf*** enthält die Implementierung für die Integration in *JSF*.
- ***mailing-module-template-model-json*** enthält die Implementierung der *JSON*-Datenobjekte in Form von *Java*-Klassen.
- ***mailing-module-template-logic-api*** enthält die Spezifikation des Vorlagenmanagements.
- ***mailing-module-template-logic-impl*** enthält die Implementierung der Spezifikation des Vorlagenmanagements.
- ***mailing-module-integration-clevercure-web*** enthält die Implementierung der Integration für die Anwendung *CleverWeb*.
- ***mailing-testsuite-cdi*** enthält die Ressourcen aller Tests, die in einer *CDI*-Umgebung lauffähig sein müssen.
- ***mailing-demo-logic*** enthält die Implementierung der Geschäftslogik der Beispielanwendung.
- ***mailing-demo-web*** enthält die *Web*-Anwendung der Beispielanwendung.
- ***mailing-data-api*** enthält die Spezifikation der Geschäftslogik, die die Persistenz der *E-Mail*-Vorlagen behandeln. Es enthält auch die Datenbankzugriffsklassen in Form von *Data-Repository*-Schnittstellen.
- ***mailing-data-impl*** enthält die Implementierung der Spezifikation der Geschäftslogik.

## 4.1 Implementierung der Spezifikation

Dieser Abschnitt behandelt die Implementierung der Spezifikation, die in Kapitel 3 vorgestellt wurde.

### 4.1.1 Implementierungen für *CKEditor*

Wie im Abschnitt 3.2.1 vorgegeben, wurde ein *Plugin* in *TypeScript* implementiert, das innerhalb des *CKEditors* die Variablen verwaltet. Die Implementierung des *Plugins* in *TypeScript* war möglich, da für den *Editor CKEditor* von dem *Microsoft* verwalteten *Open-Source*-Projekt *Definitely-Typed* Typinformationen für *TypeScript* bereitgestellt werden, welche die *JavaScript*-Schnittstellen als *TypeScript*-Schnittstellen definieren. Wären keine Typinformationen zur Verfügung gestanden, hätte man die Typinformationen selber implementieren müssen, was einen erheblichen Mehraufwand bedeutet hätte.

#### *CKEditor-Plugin* in *TypeScript*

Das Variablenmanagement ist unabhängig vom verwendeten *Editor* und wurde daher vom *CKEditor-Plugin* logisch und physisch getrennt, wobei das Variablenmanagement im *TypeScript*-Modul *cc.variables* und das *CKEditor-Plugin* im *TypeScript*-Modul *cc.ckeditor.plugins* implementiert wurden.

Die voneinander getrennten *TypeScript*-Quelltextdateien werden beim Kompilieren in eine einzige *JavaScript*-Quelltextdatei zusammengeführt. Mit der Organisation in eigenen *TypeScript*-Modulen wird sichergestellt, dass nur explizit nach außen sichtbar gemachte (*export MyType {...}*) Funktionen oder Typen außerhalb des Moduls referenziert werden können. Ein *TypeScript*-Modul wird in ein korrespondierendes *JavaScript*-Modul übersetzt. Die Verwendung von Modulen bringt auch den Vorteil, dass am *Window*-Objekt nur das Objekt der Wurzel des Namensraums *cc* gebunden ist, wodurch das *Window*-Objekt nicht mit den eigenen *JavaScript*-Objekten "verschmutzt" wird. Die Quelltexte 4.1 und 4.2 zeigen ein *TypeScript*-Modul und das daraus resultierende *JavaScript*-Modul.

**Quelltext 4.1:** Das *TypeScript*-Modul

```
module cc.ckeditor.plugins {  
  export module variables {  
    export interface VariableMapping {  
      id:string  
    }  
  }  
}
```

**Quelltext 4.2:** Das *JavaScript*-Modul

```
var cc;  
(function (cc) {  
  var variables;  
  (function (variables_1) {  
    // The interface VariableMapping is not part of the generated JavaScript  
  })(variables = cc.variables || (cc.variables = {}));  
})(cc || (cc = {}));
```

Die *TypeScript*-Schnittstelle *VariableMapping* aus dem Quelltext 4.1 ist nicht Teil des generierten *JavaScript*-Moduls, da diese Schnittstelle nur eine Typinformation für *TypeScript* darstellt. Wäre die Schnittstelle *VariableMapping* eine *TypeScript*-Klasse, dann wäre diese Klasse auch Teil des generierten *JavaScript*-Moduls und würde als *JavaScript*-Funktion abgebildet werden.

[Microsoft 2016] beschreibt einleitend die *TypeScript*-Schnittstellen in der Dokumentation von *TypeScript* wie folgt.

*One of TypeScript's core principles is that type-checking focuses on the shape that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.*

Das Variablenmanagement in *TypeScript* ist verantwortlich für die Browserseitige Registrierung der Variablen und stellt Hilfsmethoden zur Verfügung, mit denen Variablen in der *HTML*-Vorlage gefunden und konvertiert werden können. Der Quelltext 4.3 zeigt mehrere Möglichkeiten, wie eine Variable in *TypeScript* konvertiert werden kann.



**Quelltext 4.3:** Die Variablenkonvertierung in *TypeScript*

```

1 // Helper class for the variable conversion
2 class VariableUtils {
3     private variables:VariableMapping[] = [];
4
5     // public function for the variable conversion
6     public convert(converter:(item:VariableMapping) => any
7                     = (item:VariableMapping)=> item):any[] {
8         var converted:any[] = [];
9         for (var i = 0; i < this.variables.length; i++) {
10             converted[i] = converter(this.variables[i]);
11         }
12         return converted;
13     }
14 }
15
16 // Separate class for the variable conversion
17 class MyConverter {
18     // public function for the variable conversion
19     public convert(v:VariableMapping): any {
20         return v.displayName;
21     }
22 }
23
24 // Create objects for the defined types
25 var util :VariableUtils = new VariableUtils();
26 var converter:MyConverter = new MyConverter();
27
28 // Conversion via an arrow function
29 util.convert((v:VariableMapping) => v.displayName);
30
31 // Conversion via an anonymous function
32 util.convert(function (v:VariableMapping) {
33     return v.displayName;
34 });
35
36 // Conversion via a referenced function
37 util.convert(converter.convert);

```

Die Funktion *convert* der Klasse *VariableUtils* aus dem Quelltext 4.3 definiert den Formalparameter *converter* als eine *Arrow*-Funktion, welche die Signatur der Funktion für die Konvertierung definiert und eine Standardimplementierung definiert, die verwendet wird, wenn bei der Aktivierung der Funktion *convert* für den Formalparameter *converter* kein Aktualparameter bereitgestellt wird. Eine *Arrow*-Funktion ähnelt einem *Lambda*-Ausdruck in Java. Der Typ *any[]* ist vergleichbar mit dem Datentyp *dynamic* aus C# und gibt an, dass jeder Datentyp als Typ des zurückgelieferten *Arrays* erlaubt ist.

### Variablenrepräsentation in *TypeScript*

Eine Variable wird *Java*-seitig als Objekte des Datentyps *VariableContract* abgebildet, und muss für das *JavaScript*-seitige Variablenmanagement in eine *JSON*-Zeichenkette überführt werden, die als *JavaScript*-Objekt innerhalb von *JavaScript* verwendet wird. Dafür wurde in *TypeScript* die Schnittstelle *VariableMapping* aus dem Quelltext 4.4 definiert, welche den Kontrakt einer Variable innerhalb von *TypeScript* definiert.

**Quelltext 4.4:** Die *TypeScript*-Schnittstelle *VariableMapping*

```
1 interface VariableMapping {  
2     id    :string,  
3     label:string,  
4     info  :string  
5 }
```

Die Schnittstelle *VariableMapping* ist Teil des Moduls *cc.variables*, wird mit dem *TypeScript*-Schlüsselwort *export* nach außen offengelegt und kann über den vollständigen Pfad *cc.variables.VariableMapping* innerhalb von *TypeScript* referenziert werden. Mit der Schnittstelle *VariableMapping* werden Typinformationen für die Variablenrepräsentation in *TypeScript* bereitgestellt, damit innerhalb von *TypeScript* die Typsicherheit sichergestellt werden kann.

### Variablenrepräsentation in *Java*

Die Klasse *VariableJson* aus dem Quelltext 4.5 zeigt die korrespondierende *Java*-Implementierung der Variablenrepräsentation. Mit der Klasse *VariableJson* wird sichergestellt, dass die Variablenrepräsentation in *Java* korrespondierend zur Variablenrepräsentation in *TypeScript* ist.

Die Klasse *VariableJson* stellt die Schnittstelle der Variablen, die in *Java* über die Schnittstelle *VariableContract* abgebildet sind, zu *TypeScript* bzw. *JavaScript* dar. Als *JSON-Provider* wird die Bibliothek *FasterXML-Jackson-JSON* [FasterXML 2016], vormals *Jackson-JSON* genannt, verwendet, die es erlaubt mit Annotationen deklarativ Attribute und/oder Methoden einer Klasse auf *JSON*-Attribute abzubilden. Durch den deklarativen Ansatz über Annotationen sind die annotierten Attribute und/oder die annotierten Methoden einer Klasse entkoppelt von der Repräsentation in *JSON* und können daher abgeändert werden ohne die Abbildung auf *JSON* zu beeinflussen. Nur ein Ändern des Datentyps eines Attributes oder des Rückgabewerts einer Methode wird zu Problemen führen.

Quelltext 4.5: Die Klasse *VariableJson*

```
1 @JsonTypeName(value = "variable-json")
2 public class VariableJson extends AbstractJsonModel {
3
4     private String id;
5     private String label;
6     private String info;
7
8     public VariableJson() {
9     }
10
11     public VariableJson(String id,
12                         String label,
13                         String info) {
14         this.id = id;
15         this.label = label;
16         this.info = info;
17     }
18
19     @JsonGetter("id")
20     public String getId() {
21         return id;
22     }
23
24     @JsonSetter("id")
25     public void setId(String id) {
26         this.id = id;
27     }
28
29     @JsonGetter("label")
30     public String getLabel() {
31         return label;
32     }
33
34     @JsonSetter("label")
35     public void setLabel(String label) {
36         this.label = label;
37     }
38
39     @JsonGetter("info")
40     public String getInfo() {
41         return info;
42     }
43
44     @JsonSetter("info")
45     public void setInfo(String info) {
46         this.info = info;
47     }
48
49 }
```

### Registrierung des *Plugins* im *CKEditor*

Das *Plugin* wird über eine *JavaScript*-Datei im *CKEditor* registriert, wobei folgende Konventionen eingehalten werden müssen.

- *ckeditor/plugins* ist das Verzeichnis, in dem das *Plugin* enthalten sein muss.
- *variables* ist das Verzeichnis unterhalb des Verzeichnisses *ckeditor/plugins*, in dem die *Plugin*-Ressourcen enthalten sein müssen und das den gleichen Namen haben muss wie das *Plugin*.
- *plugin.js* ist die *JavaScript*-Datei, die im Verzeichnis *ckeditor/plugins/variables* liegen muss und das implementierte *Plugin* ist.

Der Quelltext 4.6 zeigt einen Auszug aus der *JavaScript*-Datei, mit der das *Plugin* registriert wird und auch Einstellungen am *CKEditor* vorgenommen werden können. Das *Plugin* wird vom *CKEditor* nach der Initialisierung des *Editors* geladen und registriert.

**Quelltext 4.6:** Die Konfigurationsdatei für den *CKEditor*

```
1 CKEDITOR.editorConfig = function (config) {
2     // Register plugin variables
3     config.extraPlugins = "variables";
4 }
```

### Integration des *Plugins* im *CKEditor*

Die Abbildung 4.2 zeigt die Funktionsleiste des *CKEditors*, in die der rot markierte *Button* über das *Plugin* eingefügt wurde. Durch einen Klick auf diesen *Button* wird ein Dialog geöffnet, über den die zur Verfügung stehenden Variablen ausgewählt werden können.



**Abbildung 4.2:** Die *CKEditor*-Funktionsleiste

Die Abbildung 4.3 zeigt den Dialog, der vom *CKEditor-Plugin* erstellt wurde. Im Dialog stehen alle registrierten Variablen zur Auswahl. Die Bezeichnung der ausgewählten Variable ist der Text in der Auswahlkomponente und die Beschreibung ist der Text, der unterhalb der Auswahlkomponente angezeigt wird. Durch einen Klick auf den *Button OK* wird die Variable in die

Vorlage eingefügt und der Dialog geschlossen.

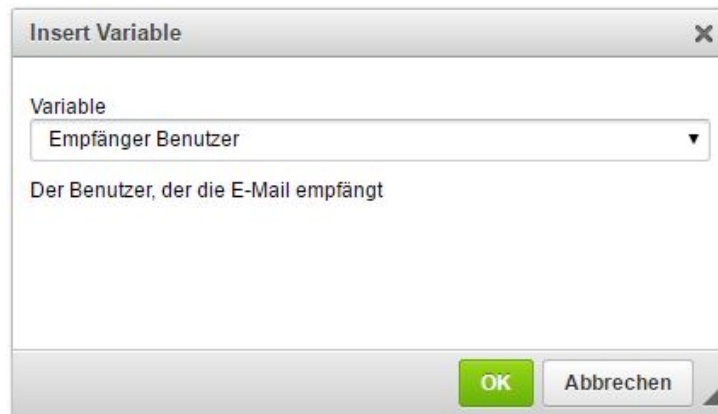


Abbildung 4.3: Der *CKEditor* Dialog für die Variablenauswahl

Die Abbildung 4.4 zeigt eine Vorlage innerhalb des *CKEditors*, wobei die eingefügten Variablen hervorgehoben werden. Die Bezeichnung der Variable stellt den Namen für den *HTML-Tag* bereit und die Beschreibung dessen Titel. Die eingefügten *HTML-Tags* dürfen nicht verändert werden, daher ist *Drag, Drop* und das Selektieren des eingefügten *HTML-Tags* nicht erlaubt, da dadurch der eingefügte *HTML-Tag* zerstört werden könnte und die Variablen nicht mehr gefunden werden können.

Sehr geehrte(r) Frau/Herr **Empfänger Benutzer**.

Hiermit informieren wir Sie über die Statusänderung bezüglich **Thema**.

Status: **Status**

Sender: **Sender Benutzer**

Abbildung 4.4: Beispiel einer Vorlage im *CKEditor*

#### 4.1.2 Implementierungen für *CDI*

Dieser Abschnitt behandelt die Implementierung für die Integration in eine *CDI*-Umgebung, wie in Abschnitt 3.2.2 beschrieben. Die Variablen und Typen der Schnittstelle *VariableResolverFactory* werden über eine *CDI*-Erweiterung gefunden, registriert und es werden die folgenden Ressourcen kontextabhängig über einen implementierten *CDI*-Erzeuger zur Verfügung gestellt:

- Das Objekt des Datentyps *VariableConfiguration* verwaltet die registrierten Variablen.
- Die Objekte des Datentyps *TemplateDataJsonBuilder* erstellen das *JSON*-Datenobjekt für eine Vorlage und eine spezifische *Template-Engine*.
- Die Objekte des Datentyps *TemplateProcessor* verwalten die Variablen in den Vorlagen.
- Objekte des Datentyps *AbstractTemplateMetadata* halten die Metadaten der Vorlagen und werden spezifisch für eine *Template-Engine* erstellt.
- Das Objekt der Klasse *CdiTemplateUtil* konvertiert die registrierten Variablen, die Objekte des Datentyps *VariableContract* sind, in Objekte der Klasse *VariableJson*, wobei die Bezeichnung und die Beschreibung sprachspezifisch ermittelt werden.

#### Vorlagenmanagement *CDI*-Erweiterung

Die implementierte *CDI*-Erweiterung *TemplateCdiExtension* hält die auffindbaren Ressourcen über die Lebensdauer der *CDI*-Umgebung persistent. Eine *CDI*-Erweiterung muss folgende Voraussetzungen erfüllen, um geladen und verwendet werden zu können:

1. Sie muss die Schnittstelle *javax.enterprise.inject.spi.Extension* implementieren,
2. in einer Datei namens *javax.enterprise.inject.spi.Extension*, die im Verzeichnis *META-INF/services* liegen muss, mit ihrem voll qualifizierten Namen registriert werden und
3. das Artefakt, dass die *CDI*-Erweiterung enthält, muss eine Datei namens *beans.xml* im Verzeichnis *META-INF* enthalten.

Die *CDI*-Erweiterung wird beim Start der *CDI*-Umgebung über den Mechanismus *Service-Provider-Interface* (*SPI*) geladen und ein Objekt der Klasse der *CDI*-Erweiterung erstellt. Dann kann das Objekt der *CDI*-Erweiterung auf Ereignisse des Lebenszyklus der *CDI*-Umgebung reagieren, in dem die *CDI*-Erweiterung Beobachtermethoden für die einzelnen Ereignisse implementiert, wie z.B.:

- *BeforeBeanDiscovery* ist das Ereignis, das einmalig beim Start der *CDI*-Umgebung ausgelöst wird, bevor Typen, *Beans* oder Injektionspunkte gesucht werden,
- *ProcessAnnotatedType* ist das Ereignis, das für jeden gefundenen injizierbaren Typ ausgelöst wird und
- *AfterBeanDiscovery* ist das Ereignis, das einmalig ausgelöst wird, wenn

alle Typen, *Beans* und Injektionspunkte gefunden und behandelt wurden.

Das erstellte Objekt der *CDI*-Erweiterung ist an sich kein *CDI-Bean*, da das Objekt der *CDI*-Erweiterung bereits existiert, bevor die *CDI*-Umgebung vollständig gestartet wurde, ist aber trotzdem in andere *CDI-Beans* injizierbar. Alle anderen *CDI-Beans* können erst nach dem erfolgreichen Start der *CDI*-Erweiterung injiziert werden.

Der Quelltext 4.7 zeigt einen Auszug aus der implementierten Klasse *TemplateCdiExtension* und zeigt die Beobachtermethoden, die auf Lebenszykluseignisse der *CDI*-Umgebung reagieren. Die *CDI*-Erweiterung *TemplateCdiExtension* findet

- alle implementierten Typen der Schnittstelle *VariableContract*, die mit der Annotation *CdiVariableContract* annotiert sind und
- alle implementierten Typen der Schnittstelle *VariableResolverFactory*, die mit der Annotation *CdiVariableResolverFactory* annotiert sind.

Die gefunden Typen werden in der *CDI*-Erweiterung registriert und über die Lebensdauer der *CDI*-Umgebung verwaltet. Bezüglich der Typen der Schnittstelle *VariableContract* sei angemerkt, dass zur Zeit nur implementierte Typen vom Typ *Enum* gefunden werden können. Alle Typen der Schnittstelle *VariableContract*, die nicht ein Aufzählungstyp sind verursachen einen Fehler und verhindern einen erfolgreichen Start der *CDI*-Umgebung. Die Variablen könnten auch über implementierte Klassen der Schnittstelle *VariableContract* definiert werden und bei ihrer Verwendung dynamisch aus der *CDI*-Umgebung geholt werden, was zur Zeit nicht benötigt wird.

Eine *CDI*-Erweiterung ist eine injizierbare Ressource, die in jedes *CDI-Bean* injiziert werden könnte, obwohl nur das Variablenmanagement sich das Objekt der *CDI*-Erweiterung injizieren sollte. Es kann nicht verhindert werden, dass sich andere *CDI-Beans* das Objekt der *CDI*-Erweiterung injizieren lassen, da eine *CDI*-Erweiterung öffentlich deklariert werden muss.

**Quelltext 4.7:** Auszug aus der *CDI-Erweiterung TemplateCdiExtension*

```

1 public class TemplateCdiExtension implements Extension {
2
3     private TemplateConfiguration templateConfig;
4     private Map<Class<? extends VariableContract>,
5               Class<VariableResolverFactory>>
6               variableResolverFactoryMap;
7
8     void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) { ... }
9
10    <T> void processCdiVariableContracts(
11          @Observes @WithAnnotations({BaseName.class,
12                                     CdiVariableContract.class})
13          ProcessAnnotatedType<T> pat) { ... }
14
15    <T> void processVariableResolverFactories(
16          @Observes @WithAnnotations(CdiVariableResolverFactory.class)
17          ProcessAnnotatedType<T> pat) { ... }
18
19 }

```

Mit der folgenden Auflistung werden die implementierten Beobachtermethoden und deren Funktionsweise erklärt:

- *beforeBeanDiscovery* ist die Beobachtermethode, die alle Objekte erstellt, welche die gefundenen Typen über die Lebensdauer der *CDI*-Umgebung verwalten,
- *processCdiVariableContracts* ist die Beobachtermethode, die die gefundenen Typen der Schnittstelle *VariableContract* behandelt und
- *processVariableResolverFactories* ist die Beobachtermethode, die die gefundenen Typen der Schnittstelle *VariableResolverFactory* behandelt.

### Vorlagenmanagement *CDI*-Erzeuger

Der implementierte *CDI*-Erzeuger *TemplateResourceProducer* produziert die kontextabhängigen Ressourcen des Vorlagenmanagements. Die Klasse *TemplateResourceProducer* ist die einzige Klasse, in die das Objekt der *CDI*-Erweiterung *TemplateCdiExtension* injiziert wird.

Im Kapitel 3 wurde vorgegeben, dass mehrere *Template-Engines* unterstützt werden müssen, daher wurde die Annotation *@FreemarkerTemplate* eingeführt, die einen Injektionspunkt für die *Template-Engine* *Freemarker* qualifiziert. In einer *CDI*-Umgebung wird ein Qualifizierer benötigt, wenn für



eine Schnittstelle mehrere Implementierungen zur Verfügung stehen, da ansonsten nicht entschieden werden kann, welche Implementierung verwendet werden soll. Für den Fall, dass es mehrere Implementierungen für eine Schnittstelle und nicht qualifizierte Injektionspunkte für diese Schnittstelle gibt, wird die Ausnahme *AmbiguousResolutionException* ausgelöst und die *CDI*-Umgebung kann nicht gestartet werden.

Es wurden jeweils eine Erzeugermethode für den Qualifizierer *@Default* und den Qualifizierer *@FreemarkerTemplate* implementiert, womit nicht qualifizierte sowie qualifizierte Injektionspunkte versorgt werden können. Für die Erzeugermethode für den Qualifizierer *@Default* wird die Implementierung für den Qualifizierer *@FreemarkerTemplate* verwendet, wodurch diese Implementierung als die Standardimplementierungen fungiert. Damit setzt man sich jedoch der Gefahr aus, dass die produzierte Standardimplementierung nicht die gewollte Implementierung ist, daher ist Vorsicht geboten, wenn dieses Verhalten geändert werden sollte.

Der Quelltext 4.8 ist ein Auszug aus der Klasse *TemplateResourceProducer* und zeigt einige der implementierten Erzeugermethoden.

Die beiden Methoden *produceDefaultTemplateBuilder* und *produceFreeMarkerTemplateBuilder* produzieren Objekte des Datentyps *TemplateDataJsonBuilder* für den sogenannten Pseudo-Geltungsbereich (*@Dependent*), wobei für jeden Injektionspunkt ein neues Objekt erstellt wird. Der Lebenszyklus von *CDI-Beans*, die sich im Pseudo-Geltungsbereich befinden, wird nicht von der *CDI*-Umgebung verwaltet und die Lebensdauer eines *CDI-Beans*, das sich im Pseudo-Geltungsbereich befindet, ist gebunden an das *CDI-Bean*, das sich das *CDI-Bean* im Pseudo-Geltungsbereich injiziert hat. Die Erzeugermethoden *produceDefaultTemplateBuilder* und *produceFreeMarkerTemplateBuilder* lassen sich als Argument ein Objekt der Schnittstelle *VariableResolverFactoryProvider* injizieren, dessen Geltungsbereich für diese Methoden nicht bekannt und auch irrelevant ist.

Die Methode *produceConfiguration* produziert ein Objekt des Datentyps *VariableConfiguration*, das die registrierten Variablen enthält und von der *CDI*-Erweiterung bereitgestellt wird. Nachdem die Schnittstelle *VariableConfiguration* nur lesenden Zugriff erlaubt, wird dieses Objekt für den Gültigkeitsbereich der Anwendung produziert, also einmalig für die gesamte Lebensdauer der *CDI*-Umgebung.

Alle injizierbaren Objekte, die sich in einem normalen Geltungsbereich befinden, werden erst beim ersten Zugriff auf eine ihrer öffentlichen Methoden erzeugt und im korrespondierenden Geltungsbereich registriert. Sollte ein injizierbares Objekt niemals verwendet werden, so wird es auch niemals er-

zeugt. Dieses Verhalten ist möglich, da alle Injektionspunkte von *Proxies*, außer Injektionspunkte von *CDI-Beans* im Pseudo-Geltungsbereich, verwaltet werden, die beim Erstellen eines *CDI-Beans* in die Injektionspunkte injiziert werden und ein abgeleiteter Typ des injizierten Typs sind. Bei einem Zugriff auf eine öffentliche Methode des injizierten Objekts, wird das korrespondierende Objekt aus dem aktuellen Geltungsbereich geholt oder vorherig erstellt und im Geltungsbereich abgelegt und der Aufruf an dieses Objekt weiter delegiert.

**Quelltext 4.8:** Die Klasse *TemplateResourceProducer*

```

1 @ApplicationScoped
2 public class TemplateResourceProducer implements Serializable {
3
4     @Produces
5     @ApplicationScoped
6     @Default
7     public VariableConfiguration produceConfiguration() {
8         return extension.getVariableConfiguration();
9     }
10
11     @Produces
12     @Dependent
13     @Default
14     public TemplateDataJsonBuilder produceDefaultTemplateBuilder
15         (final @Default VariableResolverFactoryProvider factory) {
16         return produceFreeMarkerTemplateBuilder(factory);
17     }
18
19     @Produces
20     @Dependent
21     @FreemarkerTemplate
22     public TemplateDataJsonBuilder produceFreeMarkerTemplateBuilder
23         (final @Default VariableResolverFactoryProvider factory) {
24         return new FreemarkerTemplateDataJsonBuilder()
25             .withWeakMode()
26             .withVariableResolverFactoryProvider(factory);
27     }
28
29 }
```

### Vorlagenmanagement *CDI*-Hilfsklasse

Die Klasse *CdiTemplateUtil* aus dem Quelltext 4.9 wurde implementiert, um ein injizierbares *CDI-Bean* zur Verfügung zu stellen, das Hilfsmethoden für die Konvertierung der Variablen von Objekten des Datentyps *VariableContract* in Objekte der Klasse *VariableJson* und vice versa zur Verfügung stellt. Diese Implementierung hält keinen Status, daher kann dieses *CDI-Bean* in

den Geltungsbereich der Anwendung registriert werden. Die Verwendung des Objekts der Klasse *CdiTemplateUtil* ist *Thread-safe* weil

- das Objekt keinen Status hält und
- das verwendete Objekt der Klasse *TemplateConfiguration* nur lesenden Zugriff erlaubt.

Mit der Annotation *@Typed* kann man einschränken, über welche Typen ein *CDI-Bean* injizierbar ist, was hilfreich ist, wenn die Klasse eines *CDI-Beans* mehrere Schnittstellen implementiert. Die Annotation *@Typed(CdiTemplateUtil.class)* bewirkt also, dass dieses *CDI-Bean* nur über den Typ *CdiTemplateUtil* injizierbar ist.

#### Quelltext 4.9: Die Klasse *CdiTemplateUtil*

```
1 @ApplicationScoped
2 @Typed(CdiTemplateUtil.class)
3 public class CdiTemplateUtil implements Serializable {
4
5     @Inject
6     private VariableConfiguration config;
7
8     public List<VariableJson> convertContractToJsonModel(
9         final Locale locale) { ... }
10
11     public List<VariableJson> convertContractToJsonModel(
12         final Collection<VariableContract> contracts,
13         final Locale locale) { ... }
14
15     public VariableJson convertContractToJsonModel(
16         final VariableContract contract,
17         final Locale locale) { ... }
18
19     public List<VariableContract> convertJsonModelToContract(
20         final Collection<VariableJson> jsonModels) { ... }
21
22     public VariableContract convertJsonModelToContract(
23         final VariableJson jsonModel) { ... }
24
25 }
```

#### 4.1.3 Implementierungen für *JSF*

Dieser Abschnitt beschäftigt sich mit der Implementierung der Integration des Variablenmanagements in die *View*-Technologie *JSF*, wie im Abschnitt 3.2.3 vorgegeben. Es werden der implementierte *FacesConverter* und die *CKEditor*-Integration, bereitgestellt von *PrimeFaces-Extensions*, in *JSF* be-

handelt.

### **Vorlagen-*FacesConverter***

Ein *FacesConverter* ist eine Klasse für die Konvertierung in *JSF*, welche die Schnittstelle *javax.faces.convert.Converter* implementieren muss, wobei diese Schnittstelle die folgenden beiden Methoden definiert:

1. *getAsObject* ist die Methode, die den Wert des Parameters, in Form von einer Zeichenkette, in das korrespondierende *Java*-Objekt konvertiert.
2. *getAsString* ist die Methode, die ein *Java*-Objekt in eine Zeichenkette konvertiert.

Ein *FacesConverter* wird im *JSF-Framework* über die Annotation *FacesConverter("converterName")* oder einen Eintrag in der Konfigurationsdatei *faces-config.xml* registriert. Einer *JSF*-Komponente kann in *XHTML* über das Attribut *converter* ein Konverter, entweder

- über den registrierten Namen des Konverters oder
- durch Parameterbindung auf ein Attribut eines Objekts, das ein Objekt des Datentyps *javax.faces.convert.Converter* zur Verfügung stellt, zugewiesen werden.

Die gemeinsame Logik des Konverters wurde in einer abstrakten Klasse *AbstractTemplateConverter* zusammengefasst, da sich nur die konkrete Implementierung der Schnittstelle *TemplateProcessor* für die verschiedenen *Template-Engines* unterscheidet. Da keine Injektion in *JSF*-Artefakte (*JSF 2.2*) wie z.B. *FacesConverter*, *FacesValidator* oder *Component*. möglich ist, wurde die abstrakte Klasse *AbstractTemplateConverter* und die Klasse *FreemarkerTemplateConverter* implementiert, die der Konverter für die *Template-Engine Freemarker* ist.

Ab *JSF 2.3* wird in *JSF*-Artefakten Injektion zur Verfügung stehen und man könnte dann einen anderen Ansatz wählen. Die Implementierung der Klasse *FreemarkerTemplateConverter* aus dem Quelltext aus Abbildung 4.10, die von der Klasse *AbstractTemplateConverter* ableitet, setzt über den Konstruktor den zu verwendenden Qualifizierer in Form eines Annotationsliterals, mit dem über die Hilfsklasse *BeanProvider* der Bibliothek *Delta-Spike* dynamisch das benötigte *CDI-Bean* von der *CDI*-Umgebung geholt wird. Beim Erstellen eines Objekts der Klasse *FreemarkerTemplateConverter* muss ein Objekt der Klasse *java.util.Locale* übergeben werden, damit die Bezeichnung und die Beschreibung einer Variable in einer Vorlage sprachspezifisch konvertiert werden kann. Die definierte Sprache muss die Sprache sein, für welche die Vorlage erstellt wurde.

**Quelltext 4.10:** Die Klasse *FreemarkerTemplateConverter*

```

1 public class FreemarkerTemplateConverter
2         extends AbstractTemplateConverter {
3
4     public FreemarkerTemplateConverter(final Locale locale) {
5         super(new FreemarkerTemplateLiteral(), locale);
6     }
7
8 }

```

Die abstrakte Klasse *AbstractTemplateConverter* definiert die regulären Ausdrücke, um die Variablen in einer Vorlage in Form von *HTML-Tags* zu finden und zu konvertieren:

```

String tagRegex = "<span[^>]*class=\"variable\"[^>]*>[^<]*</span>";
String idRegex  = "data-variable-id=\"(\\S+)\"";

```

- *tagRegex* ist der reguläre Ausdruck, mit dem die Variablen in ihrer *HTML*-Repräsentation in einer Vorlage gefunden werden können.
- *idRegex* ist der reguläre Ausdruck, mit dem der Bezeichner einer Variable in ihrer *HTML*-Repräsentation gefunden werden kann. Der reguläre Ausdruck *idRegex* wird auf den gefundenen *HTML-Tag* einer Variable angewendet, der mit dem regulären Ausdruck *tagRegex* gefunden wurde.

Die abstrakte Klasse *AbstractTemplateConverter* definiert auch eine Vorlage in Form einer Zeichenkette, mit der die Variablen in ihre *HTML-Tag*-Repräsentation konvertiert werden können, wobei diese Vorlage unabhängig von der verwendeten *Template-Engine* ist:

```

1 String template = "<span class=\"variable\" contenteditable=\"false\" "
2         + "data-variable-id=\"{0}\" title=\"{1}\">{2}</span>";

```

Die Vorlage *template* wird mit *java.text.MessageFormat(String, Object...)* verarbeitet, wobei der Formalparameter *Object...* eine variable Argumentliste ist, über welche die Werte für die enthaltenen Parameter der Vorlage *template* bereitgestellt werden können.

### ***Primefaces-Extension für den CKEditor***

Der *Editor CKEditor* ist eine *JavaScript*-basierte Anwendung, die nur im *Browser* der BenutzerInnen läuft. Es wird eine Integration in *JSF* benötigt, damit man

- auf *AJAX-Events* reagieren kann,
- *FacesConverter* verwenden kann und
- Parameterbindungen definieren kann.

Es ist nicht trivial eine vollwertige *JSF*-Komponente zu implementieren und das Implementieren einer solchen Komponente nimmt auch viel Zeit in Anspruch. Daher wurde auf die Implementierung von *PrimeFaces-Extensions* zurückgegriffen, die bereits eine vollwertige *JSF*-Integration in Form einer *JSF*-Komponente für den *CKEditor* bereitstellt.

Der Quelltext des *CKEditors* hat eine Größe von 1,5 *Megabyte*, daher wird der Quelltext in einem separaten Artefakt zur Verfügung gestellt. Man kann auch eine eigene Implementierung des *CKEditors* zur Verfügung stellen, sofern diese Implementierung in derselben Version vorhanden ist, die von *PrimeFaces-Extensions* unterstützt wird. Der *CKEditor* ist ein umfangreicher *Editor*, den man auch eigenen Wünschen entsprechend über die Webseite <http://ckeditor.com/builder> selbst zusammenstellen kann.

Der Quelltest 4.11 zeigt die Verwendung des *CKEditors* über die *JSF*-Komponente.

#### **Quelltext 4.11:** Die Verwendung der *JSF*-Komponente für den *CKEditor*

```
1 <pe:ckeditor id="view_id"
2           widgetVar="pfEditor"
3           value="#{model.content}"
4           converter="#{viewBean.converter}"
5           contentsCss="resources/css/myStyle.css"
6           customConfig="./ckeditor-config.js">
7 </pe:ckeditor>
```

Die folgende Auflistung erklärt die definierten Attribute, der *CKEditor JSF*-Komponente:

- *id* ist das Attribut, das die eindeutige *Id* innerhalb des Namensraums, in dem sich die Komponente befindet, definiert.
- *widgetVar* ist das Attribut, das einen global eindeutigen Namen des *JavaScript*-Objekts (*Widget*) definiert, das den Zugriff auf den *CKEditor* innerhalb von *JavaScript* ermöglicht.

- *value* ist das Attribut, das die Parameterbindung des Inhalts des *CKE-Editors* zu einem *Java*-Model definiert.
- *converter* ist das Attribut, das den zu verwendeten Konverter über seinen eindeutigen Namen oder eine Parameterbindung definiert.
- *contentCss* ist das Attribut, das den Pfad für eine eigene *CSS*-Datei, für den Inhalt der Vorlage, innerhalb des *CKEditors* definiert. Die Vorlage wird innerhalb des *Editors* als eigenständige *HMTL*-Datei behandelt, die in einer *HTML-IFrame*-Komponente gehalten wird.
- *customConfig* ist das Attribut, das den Pfad zu einer eigenen Konfigurationsdatei, in Form von einer *JavaScript*-Quelltextdatei, für den *CKEditor* definiert.

## 4.2 Vorlagenmanagement-Beispielanwendung

Dieser Abschnitt beschäftigt sich mit der implementierten Beispielanwendung für das Vorlagenmanagement, welche die Verwendung des Vorlagenmanagements im Bezug auf

- die Verwendung in einer Geschäftslogik,
- die Verwendung über eine Webseite und
- die Verwendung zum Erstellen einer *E-Mail*

aufzeigen wird.

### 4.2.1 Verwendung über eine Webseite

Die Abbildungen 4.5 und 4.6 zeigen die Weboberfläche, die für die Beispielanwendung implementiert wurde. Über dieses Formular können die Vorlagen sprachspezifisch verwaltet werden. Diese Webseite kann einfach für eine Webanwendung erstellt werden. Prinzipiell kann das Vorlagenmanagement in jeder *View*-Technologie wie z.B. *JSF* oder *Java-Server-Pages (JSP)* verwendet werden.

Die Abbildung 4.5 zeigt das Formular der Webseite, über das die Vorlagen verwaltet werden können.

<b>Mailtyp</b>	<b>Test Mailtyp</b>
<b>Sprache</b>	Deutsch
<b>Vorlage</b>	Demovorlage
<b>Standardsprache *</b>	Deutsch
<b>Name *</b>	Demovorlage
<b>Beschreibung *</b>	Das ist die Demovorlage
<b>Betreff *</b>	Statusänderung
<b>Inhalt *</b>	<div><div><div><b>B</b> <b>I</b> <b>S</b> <b>I<sub>x</sub></b></div><div></div><div>Stil Format Schriftart Grö... A- A+ U</div></div><div><p>Sehr geehrte(r) Frau/Herr <b>Empfänger Benutzer</b>.</p><p>Hiermit informieren wir Sie über die Statusänderung bezüglich <b>Thema</b>.</p><p>Status: <b>Status</b></p><p>Sender: <b>Sender Benutzer</b></p></div></div>

Abbildung 4.5: Das Formular für die Verwaltung der Vorlagen

Die Abbildung 4.6 zeigt, den Teil der Webseite, der die relevanten Daten einer Vorlage anzeigt.

› Basisvorlage
› Benutzervorlage
› JSON-Datenobjekt
› Geparste Vorlage
› Vorlagenmetadaten

Abbildung 4.6: Die Anzeige der relevanten Daten einer Vorlage



### Basisvorlage

Der Quelltext 4.12 zeigt die *Freemarker*-Basisvorlage, die von allen benutzerdefinierten Vorlagen ausgeprägt wird. Sie stellt das *HTML*-Gerüst zur Verfügung, da die Benutzervorlagen nur den Inhalt innerhalb des *HTML-Tags Body* bereitstellen.

**Quelltext 4.12:** Die *Freemarker*-Basisvorlage

```
1 <#macro includeMacro templateName>
2     <#include "${templateName}" encoding="UTF-8">
3 </#macro>
4 <!DOCTYPE html>
5 <html lang="en">
6 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
7 <body>
8 <div style="margin: 10px;">
9     <div style="padding: 5px;">
10    <@includeMacro templateName="${TEMPLATE_NAME}" />
11    </div>
12    <div style="padding: 5px;">
13    <@includeMacro templateName="${FOOTER_TEMPLATE}" />
14    </div>
15 </div>
16 </body>
17 </html>
```

Die enthaltenen Variablen werden von der *Template-Engine Freemarker* durch die Vorlagen ersetzt:

- *TEMPLATE\_NAME* ist die Variable, die den Namen für die einzufügende Vorlage definiert.
- *FOOTER\_TEMPLATE* ist die Variable, die den Namen für die einzufügende Vorlage für die Fußnote des *HTML*-Dokuments definiert.

### Benutzervorlage

Der Quelltext 4.13 zeigt die *Freemarker*-Vorlage, die von den BenutzerInnen erstellt wird. Die Vorlage enthält zwar *HTML-Markup*, aber nur den Inhalt des *HTML-Tags-Body*. Sie stellt daher kein vollständiges *HTML*-Dokument dar, wofür gerade die Basisvorlage aus dem Quelltext 4.12 implementiert wurde.



### Ausgeprägte Benutzervorlage

Der Quelltext 4.15 zeigt die ausgeprägte Benutzervorlage. Die Variablen der Vorlage aus dem Quelltext 4.13 wurden durch die serialisierten Werte des *JSON*-Datenobjekts aus dem Quelltext 4.14 ersetzt.

**Quelltext 4.15:** Die ausgeprägte Benutzervorlage

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
4   <body>
5     <div style="margin: 10px;">
6       <div style="padding: 5px;">
7         <p>Sehr geehrte(r) Frau/Herr&nbsp;Hugo Maier,</p>
8
9         <p>
10           Hiermit informieren wir Sie &uuml;ber die
11           Status&auml;nderung bez&uuml;glich&nbsp;
12           <strong>BenutzerIn Status geändert.</strong>
13         </p>
14
15         <p>Status: &nbsp;Inaktiv</p>
16
17         <p>Sender:&nbsp;Thomas Herzog</p>
18
19         <p>&nbsp;</p>
20       </div>
21     <div style="padding: 5px;">
22     </div>
23 </div>
24 </body>
25 </html>

```

### Vorlagenmetadaten

Der folgende Text zeigt die Zeichenketten, welche die in Abschnitt 3.1 vorgestellte Klasse *AbstractTemplateMetadata* produziert. Diese Ausgabe ist nur für Entwicklungszwecke interessant und zeigt die aktuellen Metadaten der Vorlage aus dem Quelltext aus Abbildung 4.13.

```

=====
FreemarkerTemplateMetadata
=====
id                : 1
version           : 1
length            : 482
variables (valid) : 4
contract          : com.clevercure.mailing.demo.logic.variable.TemplateVariable

```

```

id      : cc.module.di.SENDER_USER
name    : SENDER_USER
label-key : SENDER_USER
info-key : SENDER_USER

contract : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id      : cc.module.di.STATUS
name    : STATUS
label-key : STATUS
info-key : STATUS

contract : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id      : cc.module.di.RECIPIENT_USER
name    : RECIPIENT_USER
label-key : RECIPIENT_USER
info-key : RECIPIENT_USER

contract : com.clevercure.mailing.demo.logic.variable.TemplateVariable
id      : cc.module.di.TOPIC
name    : TOPIC
label-key : TOPIC
info-key : TOPIC

variables (invalid) : 0

```

#### 4.2.2 Verwendung in einer Geschäftslogik

Der Quelltext 4.16 zeigt die Schnittstelle *Emailservice*, die spezifiziert, wie über eine Geschäftslogik *E-Mails* erstellt werden können. Folgende Auflistung erklärt die definierten Methoden der Schnittstelle *EmailService*:

- *create(EmailDTO dto)* erstellt eine *E-Mail*.
- *create(List<EmailDTO> dtos)* erstellt mehrere *E-Mails*.
- *createAfterSuccess(EmailDTO dto)* erstellt eine *E-Mail*, nach dem erfolgreichen Beenden einer Transaktion.
- *createAfterSuccess(List<EmailDTO> dtos)* erstellt mehrere *E-Mails*, nach dem erfolgreichen Beenden einer Transaktion.

**Quelltext 4.16:** Die Schnittstelle *EmailService*

```
1 public interface EmailService extends Serializable {  
2  
3     void create(EmailDTO dto);  
4  
5     void create(List<EmailDTO> dtos);  
6  
7     void createAfterSuccess(EmailDTO dto);  
8  
9     void createAfterSuccess(List<EmailDTO> dtos);  
10  
11 }
```

Der Quelltext 4.17 zeigt die Klasse *EmailServiceCdiEventImpl*, welche die Schnittstelle *EmailService* implementiert und die *E-Mails* über *CDI-Events* erstellt.

Objekte der Klasse *CreateEmailsEvent* sind *Event*-Objekte, die alle benötigten Daten für die Erstellung einer *E-Mail* halten und nach dem Auslösen eines *Events* über den *CDI-Event-Bus* verarbeitet werden.

Es werden Objekte des Datentyps *javax.enterprise.Event* injiziert, welche mit dem Datentyp *CreateEmailsEvent* typisiert sind. Die Injektionspunkte der *Events* wurden mit den im Folgenden aufgelisteten Annotationen qualifiziert:

- *@Immediate* ist die Annotation, die das injizierte *Event* für die sofortige Ausführung qualifiziert.
- *@AfterSuccess* ist die Annotation, die das injizierte *Event* für die Ausführung nach dem erfolgreichen Abschluss einer Transaktion qualifiziert.

Durch die Qualifizierung der *Event*-Injektionspunkte wird erreicht, dass verschiedene Beobachtermethoden implementiert werden können, welche die ausgelösten Events in unterschiedlichen Phasen der Transaktion behandeln.

**Quelltext 4.17:** Die Klasse *EmailServiceCdiEventImpl*

```
1 @RequestScoped
2 @Transactional(Transaction.TxType.SUPPORTS)
3 public class EmailServiceCdiEventImpl implements EmailService {
4
5     @Inject
6     @Immediate
7     private Event<CreateEmailsEvent> createImmediateEvent;
8
9     @Inject
10    @AfterSuccess
11    private Event<CreateEmailsEvent> createAfterSuccessEvent;
12
13    @Override
14    @Transactional(Transaction.TxType.REQUIRED)
15    public void create(EmailDTO dto) {
16        createImmediateEvent.fire(new CreateEmailsEvent(dto));
17    }
18
19    @Override
20    @Transactional(Transaction.TxType.REQUIRED)
21    public void create(List<EmailDTO> dtos) {
22        createImmediateEvent.fire(new CreateEmailsEvent(dtos));
23    }
24
25    @Override
26    public void createAfterSuccess(EmailDTO dto) {
27        createAfterSuccessEvent.fire(new CreateEmailsEvent(dto));
28    }
29
30    @Override
31    public void createAfterSuccess(List<EmailDTO> dtos) {
32        createAfterSuccessEvent.fire(new CreateEmailsEvent(dtos));
33    }
34
35 }
```

Der Quelltext 4.18 zeigt die Klasse *BusinessServiceImpl*, welche die Geschäftslogik simuliert, die über die Schnittstelle *EmailService* *E-Mails* erstellt. Die zu erstellende *E-Mail* wird durch ein Objekt der Klasse *EmailDTO* repräsentiert, welches alle benötigten Informationen für das Erstellen einer *E-Mail* enthält. Das Objekt des Datentyps *EmailService* wird über Injektion von der *CDI*-Umgebung bereitgestellt. Wie im Kapitel 2 vorgegeben, dürfen die Anwendungen nicht wissen, wie *E-Mails* erstellt werden, was über die Schnittstelle *EmailService* realisiert wurde. Einer Geschäftslogik ist die konkrete Implementierung der Schnittstelle *EmailService* nicht bekannt

und daher auch nicht, dass die *E-Mails* über *CDI-Events* bzw. deren Beobachtermethoden erstellt werden.

Die *E-Mails* werden innerhalb der von der Klasse *BusinessServiceImpl* geöffneten Transaktion erstellt. Es ist nicht möglich eine Transaktion in einer Beobachtermethode zu öffnen, da die *Events* immer in der Komplettierungsphase der geöffneten Transaktion behandelt werden und es keine Möglichkeit gibt, dies zu umgehen.

**Quelltext 4.18:** Die Klasse *BusinessServiceImpl*

```

1 @RequestScoped
2 @Transactional(Transaction.TxType.REQUIRED)
3 public class BusinessServiceImpl implements BusinessService {
4
5     @Inject
6     private EmailService emailService;
7
8     @Override
9     public void doBusinessEmailImmediate() {
10         emailService.create(createEmailDto());
11     }
12
13     @Override
14     public void doBusinessEmailAfterSuccess() {
15         emailService.createAfterSuccess(createEmailDto());
16     }
17
18     private EmailDTO createEmailDto() {
19         final String email = "herzog.thomas8@gmail.com";
20         final Long mailUserId = 1L;
21         final List<Long> mailTypeIds = Collections.singletonList(1L);
22         final Locale locale = Locale.US;
23         final ZoneId zone = ZoneId.systemDefault();
24         final Map<Object, Object> userData =
25             new HashMap<Object, Object>() {{
26                 put(TemplateVariable.SENDER_USER, "Thomas Herzog");
27                 put(TemplateVariable.RECIPIENT_USER, "Hugo Maier");
28                 put(TemplateVariable.TOPIC, "User status changed");
29                 put(TemplateVariable.STATUS, "Inactive");
30             }};
31         return new EmailDTO(email,
32             locale,
33             zone,
34             mailUserId,
35             userData,
36             mailTypeIds);
37     }
38
39 }

```

Folgende Auflistung erklärt die Attribute, die beim Erstellen eines Objekts der Klasse *EmailDto* angegeben werden müssen:

- *email* ist die Zeichenkette, welche die *E-Mail*-Adresse definiert.
- *mailUserId* ist der Bezeichner der internen *Mail*-BenutzerIn, welcher die *E-Mail* in der Datenbank erstellt.
- *mailTypeIds* ist die Menge der Bezeichner, welche die *Mail*-Typen repräsentieren. Jedem *Mail*-Typ ist eine Voralge zugeordnet.
- *locale* ist das Objekt der Klasse *java.util.Locale*, das die Sprache definiert.
- *zone* ist das Objekt der Klasse *java.time.ZoneId*, das die Zone für die Datums- und Zeitformatierung definiert.
- *userData* ist der assoziative Behälter, der die Benutzerdaten enthält, die bei der Ermittlung der aktuellen Werte der Variablen verwendet werden können.

In diesem Kapitel wurde die Implementierung der Spezifikation, die in Kapitel 3 vorgestellt wurde, behandelt. Das nächste Kapitel 5 beschäftigt sich mit den Tests und der Analyse der in diesem Kapitel behandelten Implementierungen.



## Kapitel 5

# Tests und Analyse

Dieses Kapitel beschäftigt sich mit den Tests und der Analyse des implementierten Vorlagenmanagements. Es gibt zwei Arten von Tests die implementiert wurden:

1. Die Tests, die nicht auf eine *CDI*-Umgebung angewiesen sind und
2. die Tests, die auf eine *CDI*-Umgebung angewiesen sind.

### 5.1 Tests

Dieser Abschnitt beschäftigt sich mit den Tests des Vorlagenmanagements. Für die Tests wurden folgende Bibliotheken verwendet:

- *JUnit4* [JUnit 2016] ist eine Bibliothek, die ein vollwertiges Test-*Framework* darstellt, mit dem wiederholbare und reproduzierbare Tests implementiert werden können.
- *DeltaSpike* [The Apache Software Foundation 2016] ist ein Projekt von der *Apache-Software-Foundation (ASF)*, die portable *CDI*-Erweiterungen in Form von Bibliotheken bereitstellt und auch eine Bibliothek für *JUnit*-Tests in einer *CDI*-Umgebung, basierend auf der Bibliothek *JUnit*.
- *H2* [H2Database 2016] ist eine Bibliothek, die eine *In-Memory*-Datenbank zur Verfügung stellt.

Alle implementierten Tests sind nicht auf einen Anwendungsserver angewiesen und sind in jeder Entwicklungsumgebung wie z.B. *Eclipse* oder *IntelliJ* und bei einem Kompilieren über das *Buildtool Maven* ausführbar.

Bezüglich der Tests, die in einer *CDI*-Umgebung lauffähig sein müssen, sei auf den Blogeintrag von [Struberg 2012] verweisen, der die Problematik der Nutzung einer *CDI*-Umgebung innerhalb der *Java-Standard-Edition (JSE)* erklärt. Als Lösungsansatz wird ein Modul der Bibliothek von *DeltaSpike* namens *ContainerControl* vorgestellt, das eine einfache Handhabung einer

*CDI*-Umgebung ermöglicht und auch bei den folgenden Tests verwendet wird.

Die Tests wurden wie folgt organisiert:

- *com.clevercure.mailing.test.\** ist das *Java*-Paket, in dem alle implementierten Tests liegen.
- *\*.[toTestClass]Tests* ist das *Java*-Paket für eine zu testende Klasse, wobei der Paketname den Namen der zu testenden Klasse mit dem Suffix "Tests" enthält.
- *[toTestMethod]Test.java* ist die Klasse für die Tests einer Methode der zu testenden Klasse.
- *test\_case* ist der Name der einzelnen Testmethoden, der angibt, was an einer Methode getestet wird.

Die vorgestellte Konvention der Tests wurde umgesetzt, sofern es möglich war, da es auch Tests gibt, die nicht mit dieser Konvention implementiert werden können.

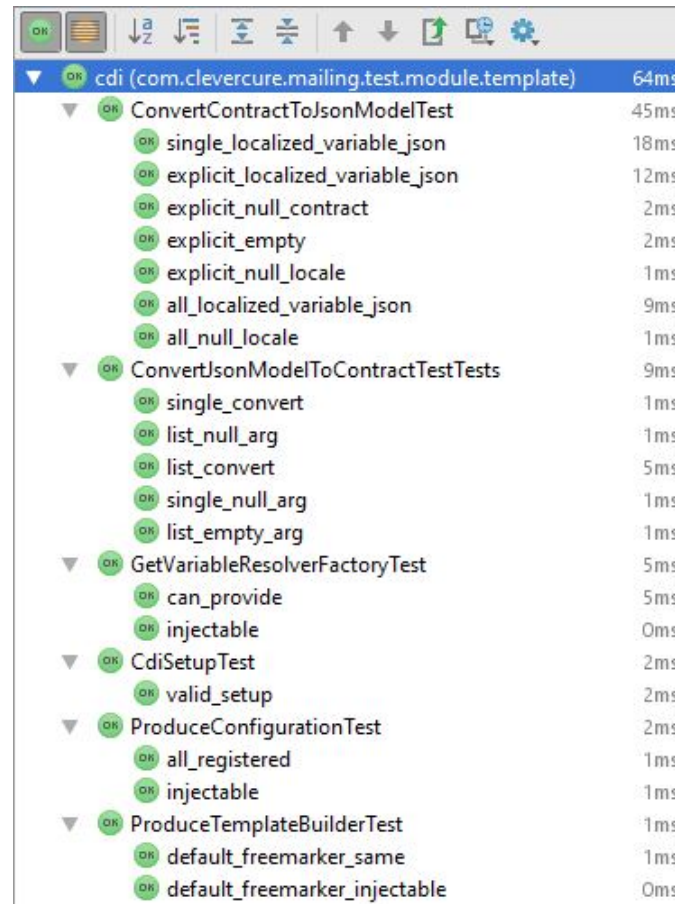
### 5.1.1 Tests der *CDI*-Integration

Die Tests aus Abbildung 5.1 testen die Implementierungen des Artefakts *mailing-moule-template-cdi*, das die *CDI*-Integration des Variablenmanagements enthält. Es werden die Klassen wie

- *TemplateCdiExtension*,
- *VariableResolverFactoryProvider*,
- *CdiTemplateUtils* und
- *TemplateResourceProducer*

getestet.

Diese Tests sind nur lauffähig in einer *CDI*-Umgebung, die mit *DeltaSpoke* im Klassenpfad gestartet werden kann. Im Klassenpfad der Tests wurden Variablen und eine Implementierung der Klasse *VariableResolverFactory* implementiert. Mit diesen Tests wird sichergestellt, dass die *CDI*-Integration des Vorlagenmanagements, aus Sicht der Implementierung, korrekt funktioniert. Diese Tests gewährleisten nicht, dass die *CDI*-Integration in jeder Implementierung einer *CDI*-Umgebung funktioniert, da es hier durchaus Unterschiede geben kann. Um garantieren zu können, dass das Vorlagenmanagement in der verwendeten implementierten *CDI*-Umgebung funktioniert, müssen Integrationstests implementiert werden, die im verwendeten Anwendungsserver ausgeführt werden.

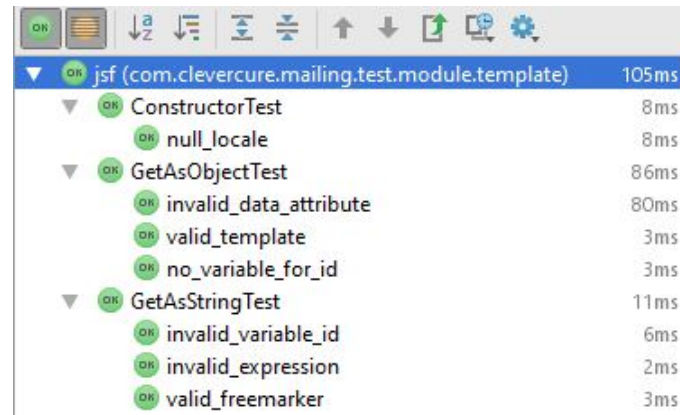


Test Name	Duration
cdi (com.clevercure.mailing.test.module.template)	64ms
ConvertContractToJsonModelTest	45ms
single_localized_variable_json	18ms
explicit_localized_variable_json	12ms
explicit_null_contract	2ms
explicit_empty	2ms
explicit_null_locale	1ms
all_localized_variable_json	9ms
all_null_locale	1ms
ConvertJsonModelToContractTestTests	9ms
single_convert	1ms
list_null_arg	1ms
list_convert	5ms
single_null_arg	1ms
list_empty_arg	1ms
GetVariableResolverFactoryTest	5ms
can_provide	5ms
injectable	0ms
CdiSetupTest	2ms
valid_setup	2ms
ProduceConfigurationTest	2ms
all_registered	1ms
injectable	1ms
ProduceTemplateBuilderTest	1ms
default_freemarker_same	1ms
default_freemarker_injectable	0ms

Abbildung 5.1: Die Tests des Artefakts *mailing-moule-template-cdi*

### 5.1.2 Tests der JSF-Integration

Die Tests aus Abbildung 5.2 testen die Implementierungen des Artefakts *mailing-module-template-jsf*, das die JSF-Integration des Variablenmanagements enthält. Es wird der implementierte *FacesConverter FreemarkerTemplateConverter* getestet. Obwohl die Klasse *FreemarkerTemplateConverter* innerhalb des *JSF-Frameworks* verwendet wird, ist es nicht notwendig, eine JSF-Umgebung zu simulieren oder zu starten. Der Konverter greift nicht auf die Formalparameter *UIComponent* und *FacesContext* zu, daher ist es nicht notwendig, *Mocks* für diese Objekte zur Verfügung zu stellen. Diese Tests sind aber auf eine *CDI*-Umgebung angewiesen, da in der Implementierung mit der *CDI*-Umgebung interagiert wird und *CDI-Beans* verwendet werden.



Test Name	Execution Time
jsf (com.clevercure.mailing.test.module.template)	105ms
ConstructorTest	8ms
null_locale	8ms
GetAsObjectTest	86ms
invalid_data_attribute	80ms
valid_template	3ms
no_variable_for_id	3ms
GetAsStringTest	11ms
invalid_variable_id	6ms
invalid_expression	2ms
valid_freemarker	3ms

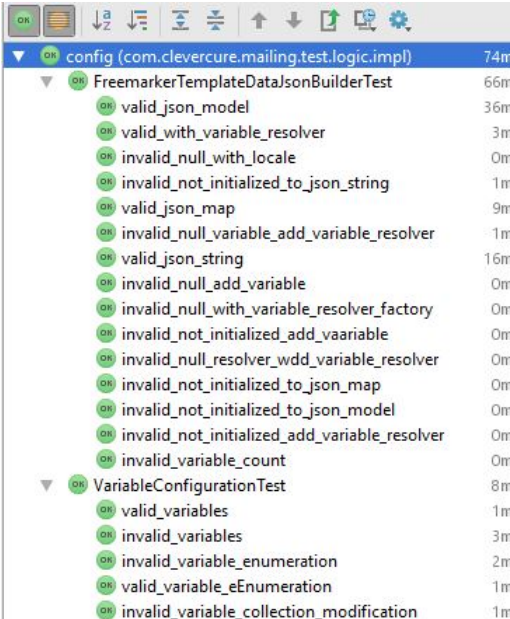
Abbildung 5.2: Die Tests des Artefakts *mailing-moule-template-jsf*

### 5.1.3 Tests des Vorlagenmanagements

Die Tests aus Abbildung 5.3 testen die Implementierungen des Artefakts *mailing-module-template-logic-impl*, welches die Implementierungen des Vorlagenmanagements enthält. Es werden die Klassen *VariableConfigurationImpl* und *FreemarkerTemplateData.JsonBuilder* getestet.

Diese Tests sind nicht abhängig von einer *CDI*-Umgebung und können mit der Bibliothek *JUnit4* alleine getestet werden. Es wird getestet ob Variablen korrekt registriert werden und in einem Objekt der Klasse *VariableConfigurationImpl* korrekt verwaltet werden und ob die Klasse *FreemarkerTemplateData.JsonBuilder* in der Lage ist die verschiedenen Repräsentationen des Datenobjekts zu produzieren, dass die Daten für eine Voralge hält.

Es müssen noch weitere Tests für die beiden Klassen *FreemarkerTemplateProcessor* und *FreemarkerTemplateMetadata* implementiert werden.



Test Name	Execution Time
config (com.clevercure.mailing.test.logic.impl)	74ms
FreemarkerTemplateDataJsonBuilderTest	66ms
valid_json_model	36ms
valid_with_variable_resolver	3ms
invalid_null_with_locale	0ms
invalid_not_initialized_to_json_string	1ms
valid_json_map	9ms
invalid_null_variable_add_variable_resolver	1ms
valid_json_string	16ms
invalid_null_add_variable	0ms
invalid_null_with_variable_resolver_factory	0ms
invalid_not_initialized_add_variable	0ms
invalid_null_resolver_wdd_variable_resolver	0ms
invalid_not_initialized_to_json_map	0ms
invalid_not_initialized_to_json_model	0ms
invalid_not_initialized_add_variable_resolver	0ms
invalid_variable_count	0ms
VariableConfigurationTest	8ms
valid_variables	1ms
invalid_variables	3ms
invalid_variable_enumeration	2ms
valid_variable_eEnumeration	1ms
invalid_variable_collection_modification	1ms

Abbildung 5.3: Die Tests des Artefakts *mailing-moule-template-logic-impl*

## 5.2 Analyse

Dieser Abschnitt beschäftigt sich mit der Analyse der erreichten Ziele des Vorlagenmanagements, dessen Integration in eine *CDI*-Umgebung und *JSF*, sowie der implementierten Beispielwebanwendung. Es wurden alle Anforderung, die im Kapitel 2 vorgegeben wurden, erfüllt.

### 5.2.1 *CKEditor-Plugin* des Vorlagenmanagements

Es wurde erfolgreich ein *Plugin* für den *CKEditor* implementiert, sowie ein Variablenmanagement für die *Browser*-seitige Verwaltung der Variablen. Wie in Abschnitt 3.2.1 vorgegeben, wurde das *CKEditor-Plugin* und das Variablenmanagement in *TypeScript* getrennt voneinander in eigenen Quelltextdateien implementiert. Die *TypeScript*-Quelltexte befinden sich zurzeit noch in der Beispielwebanwendung, da die Entwicklung in einem eigenen Projekt nicht möglich war, da das *Hot-Code-Deployment* für *Java*-Ressourcen (*src/main/resources*) nicht unterstützt wird. Die Quelltextdateien können einfach in ein anderes Projekt verschoben werden. Die Quelltextdateien werden jetzt noch über die Entwicklungsumgebung kompiliert. In Zukunft können die *TypeScript*-Quelltextdateien über das *Maven-Build-Plugin* *maven-grunt-plugin* auch automatisiert bei jedem *Maven-Build* kompiliert werden, was sehr zu empfehlen ist.

### 5.2.2 *CDI*-Integration des Vorlagenmanagements

Es wurde erfolgreich die Integration des Vorlagenmanagement in eine *CDI*-Umgebung implementiert. Die in Abschnitt 4.1.2 behandelte Integration in eine *CDI*-Umgebung, wurde über eine portierbare *CDI*-Erweiterung realisiert. Als nächster Schritt könnten auch Variablen unterstützt werden, die nicht über einen eigenen Aufzählungstyp definiert werden. Dazu müsste die Methode *processCdiVariableContracts* der Klasse *TemplateCdiExtension* und die Klasse *VariableConfigurationImpl* erweitert werden. Die Klasse *TemplateCdiExtension* müsste die registrierten Typen der Schnittstelle *VariableContract* in einem Behälter verwalten und die Klasse *VariableConfigurationImpl* müsste in der Lage sein, die registrierten Variablen dynamisch aus einer *CDI*-Umgebung zu holen. Es müsste eine Schnittstelle eingeführt werden, die das Holen der Variablen aus der *CDI*-Umgebung für die Klasse *VariableConfigurationImpl* abstrahiert, damit keine Abhängigkeiten zu Klassen von *CDI* gibt.

### 5.2.3 *JSF*-Integration des Vorlagenmanagements

Es wurde erfolgreich eine Integration in *JSF* implementiert, wobei diese Integration über den implementierten *FacesConverter FreemarkerTemplateConverter* erreicht wurde, der die Vorlagen von ihrer *HTML*-Repräsentation in die *Freemarker*-Repräsentation konvertieren kann. Wie in Abschnitt 4.1.3 vorgestellt, wurde die gemeinsame Logik in einer abstrakten Klasse *AbstractTemplateConverter* gekapselt, der nur bekanntgegeben werden muss, welche konkrete Implementierung, definiert über ein Annotationsliteral für den Qualifizierer, genutzt werden soll. Wenn man auf *JSF 2.3* wechselt, könnte man die dynamische Interaktion mit der *CDI*-Umgebung durch statische Injektionspunkte ersetzen, die auch beim Start der *CDI*-Umgebung validiert werden.

## Kapitel 6

# Zusammenfassung, weitere Aufgaben und Erfahrungen

Dieses Kapitel beschäftigt sich mit der Zusammenfassung, den weiteren Aufgaben und den gemachten Erfahrungen während der Entwicklung des Vorlagenmanagements.

### 6.1 Zusammenfassung

Dieser Abschnitt befasst sich mit der Zusammenfassung dieser Bachelorarbeit.

Bei der Implementierung des Vorlagenmanagements wurde darauf geachtet, dass die Implementierungen in eigenen Modulen organisiert werden, die entkoppelt voneinander sind und dass die Implementierungen erweiterbar sind, damit sich zukünftige Anforderungen leicht integrieren lassen. Die Strukturierung des Vorlagenmanagements in eigenen Modulen ist ein Resultat der Analyse der Anwendung *CCMail*, die in der theoretischen Bachelorarbeit durchgeführt wurde. *CCMail* ist in einem einzigen Projekt organisiert worden und kann nicht mehr erweitert werden. Durch die Modularisierung ist das Vorlagenmanagement flexibel, wie die implementierte Integration in *CDI*, *JSF* und das *Mail-DB*-Schema aufzeigt.

Das Vorlagenmanagement ist nicht auf die Verwendung in den Anwendungen innerhalb der Softwarelösung *clevercure* beschränkt, sondern kann auch in anderen Anwendungen verwendet werden, sofern die technischen Voraussetzungen erfüllt sind. Dadurch könnte das Vorlagenmanagement in Zukunft auch in Anwendungen verwendet werden, die neu für die Softwarelösung *clevercure* implementiert werden.

Das Vorlagenmanagement erfüllt alle Grundvoraussetzungen um in den An-

wendungen *CleverInterface*, *CleverWeb*, *CleverSupport* und *CleverDocument* integriert werden zu können. Die Integration in die Anwendungen *CleverWeb*, *CleverSupport* und *CleverDocument* sollte leicht realisierbar sein, da diese Anwendungen alle technischen Grundvoraussetzungen erfüllen. Die Integration in die Anwendung *CleverInterface* könnte ein Problem darstellen, da es bei *CleverInterface* Einschränkungen bezüglich den verwendeten Technologien gibt und man hier stark von der Laufzeitumgebung *IIB* und von der *IBM* abhängig ist.

Das Vorlagenmanagement ist zwar fertiggestellt, es werden sich aber sicherlich noch neue Anforderungen ergeben, die sich aber auf neue Funktionalitäten und Erweiterungen beschränken werden. Die Grundfunktionalität und die Integration in die verschiedenen Umgebungen ist fertiggestellt und kann bei Bedarf jederzeit erweitert werden.

## 6.2 Weitere Aufgaben

Dieser Abschnitt befasst sich mit den weiteren Aufgaben nach der Implementierung des Vorlagenmanagements.

Nach der Implementierung des Vorlagenmanagements muss die Integration des Vorlagenmanagements in die Anwendungen *CleverWeb*, *CleverSupport*, *CleverDocument* und *CleverInterface* implementiert werden. Die Integration in die Anwendung *CleverInterface* muss warten, bis die verwendete Laufzeitumgebung *IIB* Java 8 unterstützt. Sollte *IIB* Java 8 nicht in absehbarer Zeit unterstützen, so wird man das Vorlagenmanagement auf Java 7 migrieren müssen, was aber nicht anzuraten ist. Die Beispielwebanwendung hat aufgezeigt, wie einfach es ist, eine *JSF*-Seite für die Verwaltung von Vorlagen zu implementieren, und wie einfach *E-Mails* über eine Geschäftslogik erstellt werden können. Somit sollte sich die Integration in die Anwendungen *CleverWeb*, *CleverSupport* und *CleverDocument* einfach und schnell realisieren lassen.

Nachdem die Integration für die Anwendungen der Softwarelösung *clever-cure* implementiert wurden, muss die Anwendung *CleverMail* implementiert werden, welche die *E-Mails* versendet. Diese Entwicklung kann auch parallel zur Implementierung der Integration des Vorlagenmanagements erfolgen.

## 6.3 Erfahrungen

Dieser Abschnitt befasst sich mit den gemachten Erfahrungen während der Entwicklung des Vorlagenmanagements.



Es wahr sehr interessant zu sehen, wie leicht sich ein Softwaremodul, sofern es die Voraussetzungen erfüllt, in die verschiedensten Umgebungen integrieren lässt und wie die Interaktion zwischen den verschiedenen Umgebungen funktioniert. Die Trennung der Schichten über eigene Modellklassen, wie bei der Schnittstelle *VariableContract*, die

- über die Klasse *VariableJson* für *JSON* in *Java* und
- über die Schnittstelle *VariablenMapping* für *JavaScript* in *TypeScript*

repräsentiert wird, um die Schichten und auch die verschiedenen Technologien voneinander zu trennen, hat mir aufgezeigt, wie unabdingbar die Schichtentrennung ist. Das Vermeiden von Schichtentrennung wird aus meiner Erfahrung heraus oft mit Optimierung, Kostengründen und Ressourcenknappheit begründet. Die Schichtentrennung wird von vielen unterschätzt, aber wenn Umstrukturierungen an Modellen vorgenommen werden müssen, dann merkt man erst, wie sich die fehlende Schichtentrennungen negativ auswirkt. Meistens hat man den Fall, dass bei einer Änderung eines Modells einer höheren Schicht der gesamte Quelltext über alle Schichten hinweg Fehler aufweist.

Die Entwicklung des *CKEditor-Plugins* in *TypeScript* hat mir aufgezeigt, dass *TypeScript*, trotz aller Kritik, durchaus Zukunft hat, obwohl es auch einige Probleme mit *TypeScript* gibt, wie z.B.

- die Versionierung der Typinformationen für *JavaScript*-Bibliotheken, die nicht die Versionen der *JavaScript*-Bibliotheken widerspiegeln,
- die Organisation des *Github-Repositories* von *DefinitelyTyped*, das in einem einzigen *Repository* alle Typinformationen für alle *JavaScript*-Bibliotheken enthält und
- die rasante Weiterentwicklung von *TypeScript*, mit der man schwer mithalten kann.

Trotz aller Probleme ist es sehr angenehm, in *TypeScript* zu entwickeln und es ähnelt immer mehr der Entwicklung in einer höheren Programmiersprache wie z.B. *Java* oder *.NET*.

Beim Verfassen dieser Bachelorarbeit fiel es mir teilweise schwer, mich für einzelne Aspekte der Implementierung des Vorlagenmanagements zu entscheiden, die in dieser Bachelorarbeit behandelt wurden, da viele verschiedene Technologien, *Frameworks* und Sprachen in den Implementierungen des Vorlagenmanagements verwendet werden. Im Gegensatz zur theoretischen Bachelorarbeit, fiel es mir leichter, die praktischen Bachelorarbeit auszuarbeiten und die theoretische Bachelorarbeit war eine gute Vorbereitung für die praktische Bachelorarbeit.

# Quellenverzeichnis

## Literatur

- Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides (1994). *Design Patterns*. USA: Addison-Wesley Professional.
- Raoul-Gabriel, Urma, Mario Fusco und Alan Mycroft (2014). *Java 8 In Action*. India: Wiley India.

## Online-Quellen

- CKSource (2016). *What is CKEditor ?* URL: <http://ckeditor.com/about>.
- FasterXML, jackson (2016). *Jackson Project Home @github*. URL: <https://github.com/FasterXML/jackson>.
- Fowler, Martin (2005). *FluentInterface*. URL: <http://martinfowler.com/bliki/FluentInterface.html>.
- H2Database (2016). *H2 Database Engine*. URL: <http://www.h2database.com/html/main.html>.
- JetBrains (2016). *IntelliJ IDEA*. URL: <https://www.jetbrains.com/idea/>.
- JUnit (2016). *About JUnit4*. URL: <http://junit.org/junit4/>.
- Microsoft (2016). *Interfaces*. URL: <https://www.typescriptlang.org/docs/handbook/interfaces.html>.
- PrimeFaces-Extensions (2016). *PrimeFaces Extensions*. URL: <http://primefaces-extensions.github.io/>.
- RedHat (2016). *What is WildFly ?* URL: <http://wildfly.org/about/>.
- Struberg, Mark (2012). *Control CDI Containers in SE and EE*. URL: <https://struberg.wordpress.com/2012/03/17/controlling-cdi-containers-in-se-and-ee/>.
- The Apache Software Foundation (2016). *About DeltaSpike*. URL: <https://deltaspike.apache.org/index.html>.