



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 Titel 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

Kurzfassung	iii
Abstract	v
1 Die Einleitung	1
1.1 Das Unternehmen <i>curecomp Software Services GmbH</i>	1
1.2 Das Vorlagenmanagement für <i>CleverMail</i>	2
1.3 Die Rahmenbedingungen	3
2 Das Ziel des Projekts	4
2.1 Die funktionalen Ziele	5
2.1.1 Die Variablen für die Vorlagen	5
2.1.2 Die Mehrsprachigkeit der Variablen	5
2.1.3 Die automatische Registrierung der Variablen	5
2.1.4 Die Mehrsprachigkeit der Vorlagen	6
2.1.5 Die Persistenz der Vorlagen	6
2.1.6 Die Verwaltung der Vorlagen über eine Webseite	6
2.2 Die technischen Ziele	7
3 Das Lösungskonzept	9
3.1 Die Spezifikation des Vorlagenmanagements	9
3.1.1 Die Schnittstellen und abstrakten Klassen	10
3.2 Die Spezifikation der Vorlagenintegration	20
3.2.1 Das Vorlagenmanagement in <i>TypeScript</i>	20
3.2.2 Das Vorlagenmanagement in <i>CDI</i>	20
3.2.3 Das Vorlagenmanagement in <i>JSF</i>	21
3.2.4 Das Vorlagenmanagement in <i>Mail-DB-Schema</i>	21
4 Die Realisierung	22
4.1 Die Implementierung der Spezifikationen	24
4.1.1 Die Implementierung für <i>CKEditor</i>	24
4.1.2 Die Implementierungen für <i>CDI</i>	30
4.1.3 Die Implementierungen für <i>JSF</i>	36
4.2 Die Vorlagenmanagement Beispielanwendung	40

4.2.1	Die Verwendung über eine <i>Web</i> -Oberfläche	40
4.2.2	Die Verwendung in einer Geschäftslogik	45
5	Die Tests und erreichten Ziele	50
5.1	Die Tests	50
5.1.1	Die Tests der <i>CDI</i> -Integration	51
5.1.2	Die Tests der <i>JSF</i> -Integration	52
5.1.3	Die Tests des Vorlagenmanagements	53
5.2	Die erreichten Ziele	54
5.2.1	Das <i>CKEditor-Plugin</i> für das Vorlagenmanagement . .	55
5.2.2	Die <i>CDI</i> -Integration des Vorlagenmanagements	55
5.2.3	Das Vorlagenmanagement in <i>JSF</i>	55
5.2.4	Das Vorlagenmanagement in <i>Mail</i> -DB-Schema	56
6	Die Zusammenfassung	57
	Quellenverzeichnis	59
	Literatur	59
	Online-Quellen	59

Kurzfassung

Das implementierte 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*-Funktionen,
- Methodenreferenzen und
- die *Stream-API* hat den Quelltext lesbarer gemacht.

Die Integration des Vorlagenmanagements in eine *CDI*-Umgebung war einfach zu realisieren und hat aufgezeigt, dass jedes implementierte Softwaremodul in eine *CDI*-Umgebung verwendet werden kann. Die implementierte *CDI*-Erweiterung wird auch 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 dass die Variablen nur beim Start der *CDI*-Umgebung registriert werden und dann über die Lebensdauer der *CDI*-Umgebung unveränderlich sind.

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

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

wickelt werden sollte, ist sehr schlecht dokumentiert, daher wurde auf den *Editor CKEditor* gewechselt. Die Implementierung in *TypeScript* war die richtige Entscheidung, denn es hat die Entwicklung sehr vereinfacht, und der Quelltext ist lesbarer als der Quelltext in *JavaScript*. Für die Zukunft wird *TypeScript* noch mehr 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

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 functions,
- Method references and
- the Stream API made the source more readable.

The integration of the template management in a CDI environment was easy to realize and demonstrated, that any software module can be used in a CDI environment. The implemented CDI extension will be easy to extend and one could use more features provided by CDI. For example, producer for variables could be registered, which can 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 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 an development environment, without the need of an application server environment.

The implementation of the CKEditor plugin was proved to be easy, because the editor is documented well and there are already type information provided for TypeScript. The editor TinyMCE, which the plugin was supposed to be implemented in the first place, is not documented well, 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 of JavaScript. TypeScript will provide more language specific feature in the future, which will make the source even more easy to understand, although a migration to a newer version of TypeScript is not needed for now.

Kapitel 1

Die Einleitung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Konzeption und 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 soll 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 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 des konzipierten *Mail-Service* *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 auf Ressourcen des *Mail-Service* haben.

1.1 Das 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 *Web*-Anwendung für den webbasierten Zugriff auf *clevercure*.
- *CleverInterface*
ist eine Schnittstellenanwendung für den *XML*-basierten Datenimport und Datenexport zwischen *clevercure* und den *ERP*-Systemen der Kunden und Lieferanten.
- *CleverSupport*
ist eine unternehmensinterne *Web*-Anwendung zur Unterstützung für die 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 Das Vorlagenmanagement für *CleverMail*

Mit dem Vorlagenmanagement können Vorlagen von den EntwicklerInnen und BenutzerInnen benutzerdefiniert 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 benutzerdefinierten Vorlagen oder
- keine Möglichkeit der Nutzung von dynamischen Parametern in den Vorlagen eliminiert werden.

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

1.3 Die Rahmenbedingungen

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

Damit wird das Vorlagenmanagement mit den aktuellsten Standards und Spezifikationen implementiert. Die Funktionalität des Vorlagenmanagement muss weitestgehend ohne die Verwendung externer Bibliotheken implementiert werden. Das Vorlagenmanagement muss folgende Integrationen zur Verfügung stellen.

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

Als Entwicklungsumgebung wird *IntelliJ* verwendet, die eine bekannte Entwicklungsumgebung im *Java*-Umfeld ist und ein Produkt des Unternehmens *Jetbrains* mit Sitz in Tschechien ist. Als Applikationsserver wird *Wildfly 10.0.0*, 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

Das Ziel des Projekts

Das Ziel des Projekts Vorlagenmanagement für *Mail-Service* 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 wie z.B:

- Der *IBM-Integration-Bus (IIB)*
ist ein proprietäres Produkt des Unternehmens *IBM*, das für die *XML*-Konvertierungen und den *XML*-basierten Datenimport und Datenexport verwendet wird.
- Der Anwendungsserver *Wildfly*
ist ein zertifizierter und frei verfügbarer *JEE-7* Applikationsserver 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 dürfen. Wird eine *E-Mail*-Nachricht von einer Anwendung auf Basis einer Vorlage erstellt, so müssen die aktuellen Werte der enthaltenen Variablen der Vorlage beim Zeitpunkt des Erstellens der *E-Mail*-Nachricht ermittelt und serialisiert werden, damit die *E-Mail*-Nachricht mit dem selben 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 Datenbankmodell der Anwendung *CCMail*.

2.1 Die funktionalen Ziele

Für das Vorlagenmanagement wurden die folgende funktionalen Anforderungen definiert, die umgesetzt werden müssen.

2.1.1 Die Variablen für die Vorlagen

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

- ein 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 Die Mehrsprachigkeit der Variablen

Die zur Verfügung stehenden Variablen werden durch die EntwicklerInnen statisch definiert und müssen eine Bezeichnung und eine Beschreibung zur Verfügung stellen. Die Bezeichnung und die Beschreibung der 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 Die 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 registriert zu werden.

2.1.4 Die 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 definierte 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 Die Persistenz der Vorlagen

Die Vorlagen müssen innerhalb einer Datenbank persistent gehalten werden. Da das Vorlagenmanagement vorerst exklusiv für *CleverMail* verwendet wird, muss die Persistenz der Vorlagen innerhalb des *Mail-DB*-Schema von *CleverMail* realisiert werden. Die persistenten Vorlagen müssen versionierbar sein, damit diese von anderen Entitäten referenziert werden können, ohne dass die Gefahr besteht, dass die referenzierte Vorlage verändert wurde. Die Versionierung soll die Konsistenz der Vorlagen sicherstellen, sodass serialisierte Daten für eine Vorlage konsistent mit den enthaltenen Variablen der Vorlage sind. Vorlagen müssen explizit freigegeben werden, bevor diese verwendet dürfen. Nach einer Freigabe darf die Vorlage nicht mehr geändert werden.

2.1.6 Die 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 visa versa. Der Quelltext aus Abbildung 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 Variable wieder hergestellt werden kann.

Programm 2.1: Beispiel eines *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 aus Abbildung 2.2 zeigt das konvertierte *HTML-Markup* aus Abbildung 2.1 als *Freemarker-Vorlage*.

Programm 2.2: Konvertiertes *HTML-Markup* als *Freemarker-Template*

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

Auf der Webseite muss der *JavaScript* basierte *Editor CKEditor* verwendet werden, weil für diesen *Editor* von *PrimeFaces-Extensions* eine *JSF*-Integration, in Form einer *JSF*-Komponente, zur Verfügung gestellt wird. Es muss der *Editor CKEditor* verwendet werden, weil eine Integration in den Lebenszyklus von *JSF* notwendig ist, damit z.B. auf *AJAX-Request* reagiert werden kann, wie es in *JSF* üblich ist.

2.2 Die technischen Ziele

Dieser Abschnitt behandelt die definierten technischen Ziele, die folgend aufgelistet sind.

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

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

Kapitel 3

Das Lösungskonzept

In diesem Kapitel wird der Lösungsansatz und die Spezifikation des Vorlagenmanagements behandelt. Bei der Spezifikation handelt es sich um die Schnittstellen und die abstrakte Klassen, die die Struktur des Vorlagenmanagements definieren und 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*,
- für die *Template-Engine Velocity* oder
- für die *Template-Engine Thymeleaf*.

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

3.1 Die Spezifikation des Vorlagenmanagements

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

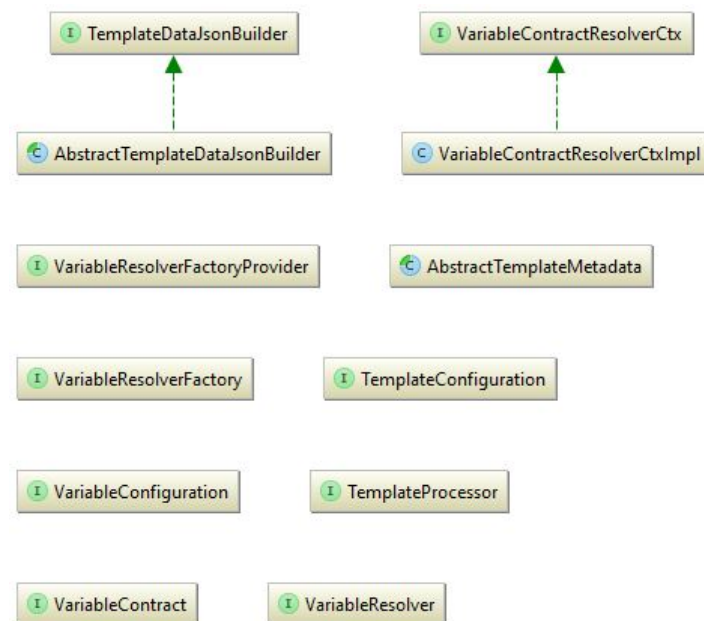


Abbildung 3.1: Klassenhierarchie des Vorlagenmanagements

3.1.1 Die Schnittstellen und abstrakten Klassen

Dieser Abschnitt behandelt die definierten Schnittstellen und abstrakten Klassen des Vorlagenmanagements. Die abstrakten Klassen implementieren die gemeinsame nutzbare Logik, die von allen konkreten Implementierungen des Vorlagenmanagements für jede *Template-Engine* genutzt werden können. Diese Spezifikationen spezifizieren 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 Metadaten der Vorlage enthält.

Die Schnittstelle *VariableContract*

Die Schnittstelle *VariableContract* aus dem Quelltext aus Abbildung 3.1 spezifiziert eine Variable, die in einer Vorlage verwendet werden kann. Objekte dieser Schnittstelle 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 Enumerationen realisiert.

siert, wobei jede Variable jeweils einen Schlüssel für die Bezeichnung und die Beschreibung bereit stellen muss.

Da es sich bei einer Variable um statische Daten handelt, also die Variablen schon zur Kompilierungszeit bekannt, ist angedacht, dass die Variablen als *Enum* implementiert werden, das die Schnittstelle *VariableContract* implementiert. Durch die Abbildung der Variablen mit einer *Enum* können mehrere Variablen in einer Klasse definiert werden, wobei eine einzelne Enumeration der *Enum* ein Objekt der Schnittstelle *VariableContract* darstellt. Alle Variablen, die mit einer *Enum* abgebildet werden, sollten demselben Modul zugeordnet sein, obwohl dies nicht zwingend erforderlich ist. Die Variablen, die mit einer *Enum* definiert wurden, werden innerhalb des Vorlagenmanagements trotzdem als einzelne Objekte der Schnittstelle *VariableContract* betrachtet. Die Tatsache dass die Variablen mit einer *Enum* abgebildet wurden, ist für das Vorlagenmanagement nur beim Registrieren der Variablen von belang und nicht bei deren weiterer Verwendung.

Eine Variable ist über seine *Id* global eindeutig identifizierbar, wobei sich die *Id* aus dem Modulnamen und den Variablennamen zusammensetzt (Bsp. *module.core.VAR_1*). Die *Id* sowie der Modulname muss sich dabei an die Namenskonvention eines *Java*-Paketnamen 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 *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.

Programm 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 }

```

Die Schnittstelle *VariableResolver*

Die Schnittstelle *VariableResolver* aus dem Quelltext aus Abbildung 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 das kontextabhängige Daten von den EntwicklerInnen bereitgestellt werden können. Durch dieses Kontextobjekt kann eine Variable in mehreren Kontexten verwendet werden und auch der aktuelle Wert einer Variable kontextabhängig ermittelt werden.

Programm 3.2: Die Schnittstelle *VariableResolver*

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

Die Schnittstelle *VariableResolver* wurde als *Functional-Interface* implementiert. Ein *Functional-Interface* ist eine Schnittstelle, die nur eine abstrakte Methode definiert, die implementiert werden muss. Eine Implementierung eines *Functional-Interface* kann über eine *Lambda*-Funktion oder Methodenreferenz bereitgestellt werden, wodurch die Notwendigkeit einer anonymen Implementierung oder der Implementierung einer Klasse für diese Schnittstelle entfällt. Die Verwendung von *Lambda*-Funktionen 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 einer *Lambda*-Funktion oder Methodenreferenz begründet ist. Die negativen Auswirkungen auf das Laufzeitverhalten können, im Bezug auf das Vorlagenmanagement, vernachlässigt werden. Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft 2014, S. 50 beschreiben den Nutzen eines *Functional-Interfaces* 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.

Die Schnittstelle *VariableResolverFactory*

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

Programm 3.3: Die Schnittstelle *VariableResolverFactory*

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

Die Schnittstelle *VariableResolver* wurde auch als *Functional-Interface* implementiert, damit Implementierungen über eine *Lambda*-Funktion oder eine Methodenreferenz bereitgestellt werden können.

Die Schnittstelle *VariableResolverFactoryProvider*

Die Schnittstelle *VariableContractFactoryProvider* aus dem Quelltext aus Abbildung 3.4 spezifiziert wie Objekte der Schnittstelle *VariableResolverFactory* produziert werden. Ein Objekt der Schnittstelle *VariableResolverFactoryProvider* kann Objekte der Schnittstelle *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 dieser Schnittstelle die Objekte der Schnittstelle *VariableResolverFactory* produziert werden können, die von der *CDI*-Umgebung zur Verfügung gestellt werden.

Programm 3.4: Die Schnittstelle *VariableResolverFactoryProvider*

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

Die Schnittstelle *VariableResolverFactoryProvider* wurde auch als *Functional-Interface* implementiert, damit Implementierungen über eine *Lambda*-Funktion oder eine Methodenreferenz bereitgestellt werden kann.

Die Schnittstelle *VariableContractResolverCtx*

Die Schnittstelle *VariableContractResolverCtx* aus dem Quelltext aus Abbildung 3.5 spezifiziert den Kontext, der bei 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 möglich die Variablen außerhalb vom Vorlagenmanagements zu verwenden.

Programm 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,  
10                        Class<T> clazz);  
11 }
```

Die Schnittstelle *TemplateProcessor*

Die Schnittstelle *TemplateProcessor* aus dem Quelltext aus Abbildung 3.7 spezifiziert wie die Variablen in einer Vorlagen behandelt werden. Objekte dieser Schnittstelle können Variablen in einer Vorlage für eine bestimmte *Template-Engine* finden und konvertieren. Ein Objekt der Schnittstelle *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 konkrete Implementierung der Schnittstelle *TemplateProcessor* ist eine Implementierung für eine bestimmte *Template-Engine*, da die in der Vorlage verwendeten Variablen in Form von Ausdrücken spezifisch für die verwendete *Template-Engine* sind.

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

Programm 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);
```

Diese Methoden definieren als Formalparameter für den benötigte Konverter ein *Functional-Interface* namens *Function*, welches von *Java 8* bereitgestellt wird. Dadurch ist das Spezifizieren einer eigenen Schnittstelle für die Konvertierung nicht mehr nötig. Der Konverter kann über eine *Lambda-Funktion* oder 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.

Programm 3.7: Die Schnittstelle *TemplateProcessor*

```
1 public interface TemplateProcessor {
2
3     String replaceExpressions(String template,
4                             Function<VariableContract, String>
5                             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 }
```

Die Schnittstelle *TemplateDataJsonBuilder*

Die Schnittstelle *TemplateDataJsonBuilder* aus dem Quelltext aus Abbildung 3.8 spezifiziert die Signatur eines *Builders*, der das Datenobjekt erstellt, welches die Daten für das Parsen einer Vorlage enthält. Die *E-Mail-Nachrichten* werden persistent gehalten, wobei nach der Erstellung einer

E-Mail-Nachricht, dessen Inhalt unveränderbar sein muss. Das Datenobjekt enthält Daten wie

- die Sprache in der die *E-Mail* versendet wird,
- 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.

Dieses Datenobjekt kann als *JSON* in den folgenden Formen vom *Builder* bereitgestellt werden.

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

Anstatt der Serialisierung der Daten könnte auch die Vorlage geparkt 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 diesem Datenobjekt kann die korrespondierende Vorlage zu jedem Zeitpunkt mit demselben Resultat wiederhergestellt werden.

Es wurde hier das *Builder*-Muster angewendet, da sich die Konfiguration des *Builders* mit einer *Fluent-API*, wie bei einem *Builder* üblich, sehr gut abbilden lässt. Die Schnittstelle *TemplateDataJsonBuilder* spezifiziert folgende Terminalmethoden.

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

Programm 3.8: Die Schnittstelle *TemplateDataJsonBuilder*

```
1 public interface TemplateDataJsonBuilder<I,  
2     M extends AbstractTemplateMetadata<I>,  
3     B extends TemplateDataJsonBuilder> extends Serializable {  
4  
5     B withWeakMode();  
6  
7     B withLocalization(Locale locale,  
8         ZoneId zoneId);  
9  
10    B withUserData(Map<Object, Object> userData);  
11  
12    B withStrictMode();  
13  
14    B withVariableResolverFactoryProvider  
15        (VariableResolverFactoryProvider factory);  
16  
17    B withVariableResolverFactory(VariableResolverFactory factory);  
18  
19    B withTemplate(M metadata);  
20  
21    void end();  
22  
23    B addVariable(VariableContract contract,  
24        Object value);  
25  
26    B addVariableResolver(VariableContract contract,  
27        VariableResolver resolver);  
28  
29    TemplateRequestJson toJsonModel();  
30  
31    String toJsonString();  
32  
33    Map<String, Object> toJsonMap();  
34 }
```

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides 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.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides 1994, S. 96 - 106 beschreiben ausführlich das Designmuster *Builder*, jedoch ohne die Verwendung einer *Fluent*-Schnittstelle, die heutzutage über den *Builder* gelegt wird, um dessen Anwendung über Punktnotation angenehmer zu ma-

chen. Mit einer *Fluent*-Schnittstelle wird die Möglichkeit geboten, einzelne Attribute des zu bauenden Objektes zu setzen, bevor es gebaut wird. Der Quelltext aus Abbildung 3.9 illustriert, wie der *Builder* mit einer *Fluent*-Schnittstelle verwendet wird.

Programm 3.9: Beispiel der Anwendung des *Builders*

```
1 builder.withStrictMode()
2     .withLocalization(localeObj, zoneIdObj)
3     .withTemplate(templateMetadataObj)
4     .withUserData(userDataMap)
5     .withVariableResolverFactoryProvider(factoryProviderObj)
6     .toJsonModel();
```

Die abstrakte Klasse *AbstractTemplateMetadata*

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

- die Anzahl der gültigen Variablen in der Vorlage,
- die Anzahl der ungültigen Variablen in der Vorlage,
- die Zeichenlänge der Vorlage,
- die eindeutige *Id* der Vorlage,
- die Version der Vorlage und
- die Vorlage selbst werden in dieser Klasse abgebildet.

Diese Metadaten sind unabhängig der verwendeten *Template-Engine* und eine konkrete Implementierung für eine spezifische *Template-Engine* kann zusätzliche Metadaten definieren. Die Metadaten werden einmalig ermittelt und sind über die Lebenszeit des Objekts unveränderbar. Wird die Vorlage geändert so muss auch ein neues Objekt der Metadaten erstellt werden.

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder*

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder* implementiert die gemeinsam nutzbare Logik, die von allen konkreten Implementierungen 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 Die Spezifikation der Vorlagenintegration

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 Integrationen in verschiedene Umgebungen und Sprachen, um die benötigte Funktionalitäten wie

- die Verwaltung der Variablen in einem *JavaScript*-basierten *CKEditor*,
- die automatische Registrierung der Variablen in einer *CDI*-Umgebung,
- die Verwaltung der Vorlagen in einer Webseite und
- die Persistenz der Vorlagen realisieren zu können.

Folgender Abschnitt behandelt die Spezifikationen der Integrationen wie in Abschnitt 2.2 vorgegeben.

3.2.1 Das Vorlagenmanagement in *TypeScript*

Wie in Abschnitt 2.1.6 vorgegeben muss der *JavaScript*-basierte *Editor CKEditor* verwendet werden, mit dem *HTML*-basierte Vorlagen in einer Webseite bearbeitet werden können. Der *CKEditor* muss angepasst werden, damit die definierten Variablen in einer Vorlage verwendet werden können. Es wird ein *CKEditor-Plugin* in *TypeScript* entwickelt, das 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, dessen Management in einer eigenen Quelltextdatei implementiert wird, da es unabhängig von *CKEditor-Plugin* ist und daher auch anderweitig verwendet werden kann. Damit ist das Variablenmanagement entkoppelt vom *CKEditor-Plugin*.

3.2.2 Das 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 sollen Ressourcen des Vorlagenmanagements wie z.B

- Objekte der Schnittstelle *VariableResolver*,

- Objekte der Schnittstelle *VariableResolverFactory* oder
- Objekte der Schnittstelle *TemplateDataJsonBuilder* kontextabhängig zur Verfügung gestellt werden.

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

3.2.3 Das Vorlagenmanagement in *JSF*

Für die Verwaltung der Vorlagen wird eine *JSF*-Webseite implementiert. Über diese Webseite können Vorlagen erstellt, modifiziert und gelöscht werden können. 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 *HTML*-Repräsentation in die *Template-Engine* spezifische Repräsentation und visa versa übernimmt.

3.2.4 Das Vorlagenmanagement in *Mail-DB-Schema*

Eine Vorlage wird durch eine Zeichenkette repräsentiert, die innerhalb des *Mail-DB-Schema* sprachspezifisch persistent gehalten wird. Es ist nicht erforderlich eine eigene Tabellenstruktur für die Vorlagen zu definieren um es von den *Mail*-Tabellen zu abstrahieren, da die Vorlagen einen essentiellen Teil von *CleverMail* darstellen und daher auch die Vorlagen bzw. deren persistente Repräsentation voll in das *Mail-DB-Schema* integriert werden müssen. Sollten die Vorlagen außerhalb von *CleverMail* verwendet werden so kann dies leicht realisiert werden, da eine Vorlage nur eine Zeichenkette darstellt, die einfach persistent gehalten werden kann.

Kapitel 4

Die 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 folgenden Projektstruktur organisiert wurden.

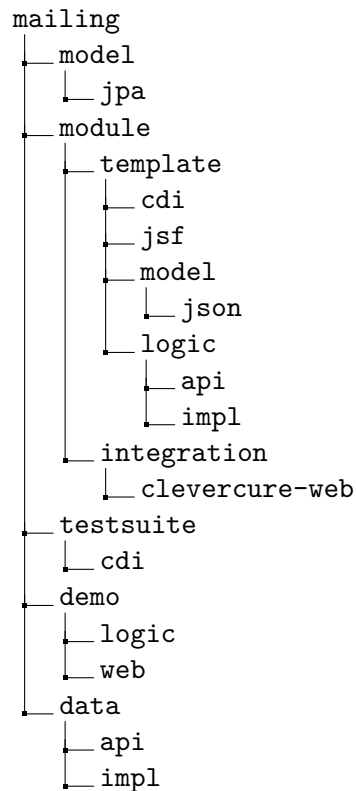


Abbildung 4.1: Verzeichnisstruktur der *Maven*-Projekte

Das *Maven*-Wurzelprojekt *mailing* organisiert die Metadaten wie die EntwicklerInnen, die an diesem Projekt mitwirken, 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 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 (Spezifikationen) vollständig getrennt von deren Implementierungen. Folgende Auflistung beschreibt alle konkreten Artefakte (*Java-Archive*), die aus dem Wurzelprojekt *mailing* erstellt werden können:

- ***mailing-model-jpa***
ist das Artefakt, das die Klassen mit den *JPA*--Entitäten enthält, die die Datenbank in *Java* abbilden.
- ***mailing-module-template-cdi***
ist das Artefakt, das die Implementierung der *CDI*-Erweiterung für die Integration in eine *CDI*-Umgebung enthält.
- ***mailing-module-template-jsf***
ist das Artefakt, das die Implementierung für die Integration in *JSF* enthält.
- ***mailing-module-template-model-json***
ist das Artefakt, das die Implementierung der *JSON*-Datenobjekte in Form von *Java*-Klassen enthält.
- ***mailing-module-template-logic-api***
ist das Artefakt, das die Spezifikation des Vorlagenmanagements enthält.
- ***mailing-module-template-logic-impl***
ist das Artefakt, das die Implementierung der Spezifikation des Vorlagenmanagements enthält.
- ***mailing-module-integration-clevercure-web***
ist das Artefakt, das die Implementierung der Integration für die Anwendung *CleverWeb* enthält.
- ***mailing-testsuite-cdi***,
ist das Artefakt, das die Ressourcen aller Tests, die in einer *CDI*-Umgebung lauffähig sein müssen, enthält.
- ***mailing-demo-logic***
ist das Artefakt, das die Schicht der Geschäftslogik der Beispielanwendung enthält.
- ***mailing-demo-web***

ist das Artefakt, das die *Web*-Anwendung der Beispielanwendung enthält.

- ***mailing-data-api***
ist das Artefakt, das die Spezifikation der Geschäftslogik enthält, die die Persistenz der *E-Mail*-Vorlagen behandeln. Es enthält auch die Datenbankzugriffsklassen in Form von *Data-Repository*-Schnittstellen.
- ***mailing-data-impl***
ist das Artefakt, das die Implementierung der Geschäftslogik enthält.

4.1 Die Implementierung der Spezifikationen

Dieser Abschnitt behandelt die Implementierungen der Spezifikationen, die im Kapitel 3 vorgestellt wurden.

4.1.1 Die Implementierung 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* vom dem, von *Microsoft* verwalteten, *Open-Source* Projekt *DefinitelyTyped* Typinformationen für *TypeScript* bereitgestellt werden, die die *JavaScript*-Schnittstellen als *TypeScript*-Schnittstellen definieren. Hätten keine Typinformationen zur Verfügung gestanden, hätte man die Typinformationen selber implementieren müssen, was einen erheblichen Mehraufwand bedeutet hätte.

Das *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 aus den Abbildungen 4.1 und 4.2 zeigen ein *TypeScript*-Modul und das daraus resultierende *JavaScript*-Modul.

Programm 4.1: Das *TypeScript*-Modul

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

Programm 4.2: Das *JavaScript*-Modul

```
var cc;  
(function (cc) {  
  var variables;  
  (function (variables_1) {  
    // VariableMapping nicht Teil des generierten JavaScripts  
  })(variables = cc.variables || (cc.variables = {}));  
})(cc || (cc = {}));
```

Die *TypeScript*-Schnittstelle *VariableMapping* aus dem Quelltext aus Abbildung 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.

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

Programm 4.3: Beispiele für Variablenkonvertierungen in *TypeScript*

```

1 // Hilfsklasse für die Konvertierung der Variablen
2 class VariableUtils {
3     private variables:VariableMapping[] = [];
4
5     // Öffentliche Funktion für die Konvertierung der Variablen
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 // Eigene Klasse für die Konvertierung
17 class MyConverter {
18     // Öffentliche Funktion für die Konvertierung der Variablen
19     public convert(v:VariableMapping): any {
20         return v.displayName;
21     }
22 }
23
24 // Erstellen der Objekte aus den definierten Klassen
25 var util :VariableUtils = new VariableUtils();
26 var converter:MyConverter = new MyConverter();
27
28 // Konvertierung mit einer Arrow-Funktion
29 util.convert((v:VariableMapping) => v.displayName);
30
31 // Konvertierung mit einer anonymen Funktion
32 util.convert(function (v:VariableMapping) {
33     return v.displayName;
34 });
35
36 // Konvertierung mit einer Referenz auf eine Funktion
37 util.convert(converter.convert);

```

Die Funktion *convert* der Klasse *VariableUtil* aus dem Quelltext aus Abbildung 4.3 definiert den Formalparameter *converter* als eine *Arrow-Funktion*, die 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 einer *Lambda-Funktion* in *Java*. Der Typ *any[]* ist vergleichbar mit dem Datentyp *var* aus *.NET* und gibt an, dass jeder Datentyp als Typ des zurückgelieferten *Arrays* erlaubt ist.

Die Variablenrepräsentation in *TypeScript*

Die Variablen werden *Java*-seitig als Objekte der Schnittstelle *VariableContract* abgebildet, und müssen für das *JavaScript*-seitige Variablenmanagement in eine *JSON*-Zeichenkette überführt werden, die als *JavaScript*-Objekte innerhalb von *JavaScript* verwendet werden. Dafür wurde in *TypeScript* die Schnittstelle *VariableMapping* aus dem Quelltext aus Abbildung 4.4 definiert, die die Struktur einer Variable innerhalb von *TypeScript* definiert.

Programm 4.4: Die *Typescript*-Schnittstelle *VariableMapping*

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

Die Schnittstelle *VariableMapping* ist Teil des Moduls *cc.variables* und 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.

Die Variablenrepräsentation in *Java*

Die Klasse *VariableJson* aus dem Quelltext aus Abbildung 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 definiert über die Schnittstelle *VariableContract* zu *TypeScript* bzw. *JavaScript* dar. Als *JSON-Provider* wird die Bibliothek *FasterXML-Jackson-JSON*, 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 kann zu Problemen führen.

Programm 4.5: Die Schnittstelle *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 displayName,
13                         String tooltip) {
14         this.id = id;
15         this.label = displayName;
16         this.info = tooltip;
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("displayName")
30     public String getLabel() {
31         return label;
32     }
33
34     @JsonSetter("displayName")
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 }
```

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 dem Namen des *Plugins* entspricht.
- *plugin.js*
ist die *JavaScript*-Datei, die im Verzeichnis *ckeditor/plugins/variables* liegen muss und das implementierte *Plugin* darstellt.

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

Programm 4.6: Registrierung des *CKEditor-Plugins*

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

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* definiert und erstellt wurde. In diesem Dialog stehen alle registrierten Variablen zur Auswahl. Die Bezeichnung der Variable ist der Text in der Auswahlkomponente

und die Beschreibung der ausgewählten Variable wird unterhalb der Auswahlkomponente angezeigt. Durch den Klick auf den *Button OK* wird die Variable in die Vorlage eingefügt und der Dialog wird geschlossen.

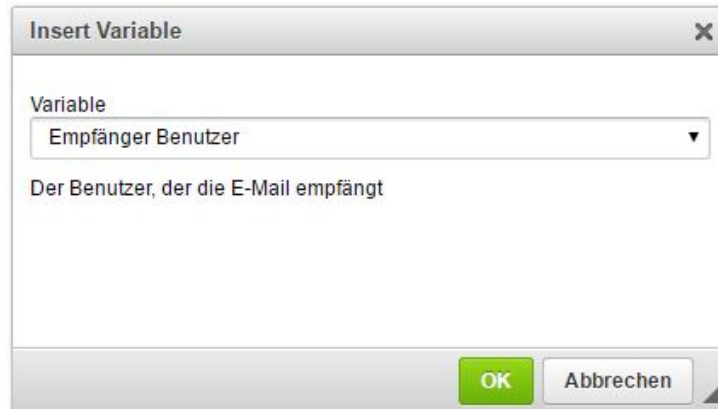


Abbildung 4.3: *CKEditor* Dialog für die Variablenauswahl

Die Abbildung 4.4 zeigt eine Vorlage innerhalb des *CKEditors*, wobei die eingefügten Variablen besonders 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 das *Drag* und *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 Die Implementierungen für *CDI*

Dieser Abschnitt behandelt die Implementierungen 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 der Schnittstelle *VariableConfiguration* ist das Objekt, dass die registrierten Variablen verwaltet.
- Die Objekte der Schnittstelle *TemplateDataJsonBuilder* sind Objekte, mit denen das *JSON*-Datenobjekt für eine Vorlage und eine spezifische *Template-Engine* erstellt werden kann.
- Die Objekte der Schnittstelle *TemplateProcessor* sind Objekte, mit denen Variablen in Vorlagen verwaltet werden können.
- Objekte der abstrakten Klasse *AbstractTemplateMetadata* sind Objekte, welche die Metadaten für Vorlagen halten und spezifisch für eine *Template-Engine* erstellt werden.
- Das Objekt der Klasse *CdiTemplateUtil* ist das Objekt mit dem die registrierten Variablen, die Objekte der Schnittstelle *VariableContract* sind, in Objekte der Klasse *VariableJson* konvertieren kann, wobei die Bezeichnung und die Beschreibung sprachspezifisch ermittelt werden.

Die Vorlagenmanagement *CDI*-Erweiterung

Die implementierte *CDI*-Erweiterung *TemplateCdiExtension* hält die auffindbaren Ressourcen über die Lebensdauer der *CDI*-Umgebung persistent. Eine *CDI*-Erweiterung für eine 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 ihren 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 aufgerufen wird bevor Typen, *Beans* oder Injektionspunkte gesucht werden.
- *ProcessAnnotatedType*
ist das Ereignis, das für jeden gefundenen injizierbaren Typ aufgerufen

wird.

- *AfterBeanDiscovery*
ist das Ereignis, das einmalig Aufgerufen 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 nachdem erfolgreichen Start der *CDI*-Erweiterung injiziert werden.

Der Quelltext aus Abbildung 4.7 zeigt einen Auszug aus der implementierten Klasse *TemplateCdiExtension* und zeigt die Beobachtermethoden, die auf Lebenszyklus Ereignisse 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 vom Typ *Enum* sind verursachen einen schweren 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.

Programm 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 }

```

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

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

Der 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. Im Fall, dass

es mehrere Implementierungen für eine Schnittstelle und nicht qualifizierte Injektionspunkte für diese Schnittstelle gibt, wird die Ausnahme *AmbiguousResolutionException* geworfen 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 Standardimplementierung 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 aus Abbildung 4.8 ist ein Auszug aus der Klasse *TemplateResourceProducer* und zeigt einige der implementierten Erzeugermethoden.

Programm 4.8: Die Klasse *TemplateResourceProducer*

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

Die beiden Methoden *produceDefaultTemplateBuilder* und *produceFreeMarkerTemplateBuilder* produzieren Objekte der Schnittstelle *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-Bean*, 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 der Schnittstelle *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 erzeugt. 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.

Die Vorlagenmanagement *CDI*-Hilfsklasse

Die Klasse *CdiTemplateUtil* aus dem Quelltext aus Abbildung 4.9 wurde implementiert, um ein injizierbares *CDI-Bean* zur Verfügung zu stellen, das Hilfsmethoden für die Konvertierung der Variablen von Objekten der Schnittstelle *VariableContract* in Objekte der Klasse *VariableJson* und visa 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.

Die Annotation `@Typed(CdiTemplateUtil.class)` bewirkt dass dieses *CDI-Bean* nur über den Typ *CdiTemplateUtil* injizierbar ist. 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.

Programm 4.9: Die Klasse *CdiTemplateUtil*

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

4.1.3 Die 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* behandelt.

Der 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.

- *getAsObject*
ist die Methode, die den Wert des Parameters, in Form von einer Zeichenkette, in das korrespondierende *Java*-Objekt konvertiert.
- *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 registrierten Namen des Konverters oder
- durch Parameterbindung auf ein Attribut eines Objekts, das ein Objekt der Schnittstelle *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* implementiert und die Klasse *FreemarkerTemplateConverter*, 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 statisch den zu verwendenden Qualifizierer in Form eines Annotationsliterals, mit dem über die Hilfsklasse *BeanProvider* der Bibliothek *DeltaSpike* dynamisch das benötigte *CDI-Beans* 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 sollte die Sprache sein, für die die Vorlage erstellt wurde.

Programm 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 }

```

Die abstrakte Klasse *AbstractTemplateConverter* definiert reguläre 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 die *Id* einer Variable, in ihrer *HTML*-Repräsentation gefunden werden kann und der reguläre Ausdruck 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.

Die *Primefaces-Extension* für den *CKEditor*

Der *Editor CKEditor* ist eine *JavaScript* basierte Anwendung, die nur am *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 ist 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.

Die Ressourcen für den *CKEditor* bewegen sich in der Größenordnung von 1,5 *Megabyte*, daher werden die Ressourcen 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 sehr umfangreicher *Editor*, den man sich auch seinen Wünschen entsprechend über die Webseite <http://ckeditor.com/builder> selbst zusammenstellen kann.

Der Quelltest aus Abbildung 4.11 zeigt die Verwendung der *JSF*-Komponente für den *CKEditor*.

Programm 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 *JSF*-Komponente für den *CKEditor*.

- *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 *CKEditors* 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, das in einer *HTML-IFrame*-Komponente gehalten wird.
- *customConfig*
ist das Attribut, das den Pfad zu einer eigenen Konfigurationsdatei, in Form von einer *JavaScript*-Datei, für den *CKEditor* definiert.

4.2 Die Vorlagenmanagement Beispielanwendung

Dieser Abschnitt beschäftigt sich mit der implementierten Beispielanwendung für das Vorlagenmanagement, welche die Verwendung des Vorlagenmanagement 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 Die Verwendung über eine *Web*-Oberfläche

Die Abbildung 4.5 zeigt die Weboberfläche, die für die Beispielanwendung implementiert wurde. Über dieses Formular können die Voralgen sprachspezifisch verwaltet werden. Diese Webseite kann einfach in jeder Webanwendung erstellt werden, daher wurde keine Implementierung zur Verfügung gestellt. Prinzipiell kann das Vorlagenmanagement in jeder *View*-Technologie wie z.B.

- *JSF* oder
- *Java-Server-Pages (JSP)* verwendet werden.

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

Abbildung 4.5: Formular für die Verwaltung der Vorlagen

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

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

Abbildung 4.6: Anzeige der aller relevanten Daten einer Vorlage

Die Dekoriovorlage

Der Quelltext aus Abbildung 4.12 zeigt die dekorierbare *Freemarker*-Vorlage, die alle Vorlagen dekorieren. Sie stellt den *HTML-Body* zur Verfügung, da die Benutzervorlagen nur den Inhalt innerhalb des *HTML-Tags Body* bereitstellen.

Programm 4.12: Die dekorierbare *Freemarker*-Vorlage

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

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

- *TEMPLATE_NAME*
ist die Variable, die den Namen für die zu einfügende Vorlage definiert und
- *FOOTER_TEMPLATE*
ist die Variable, die den Namen für die zu einfügende Vorlage für die Fußnote des *HTML*-Dokuments definiert.

Die Benutzervorlage

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

Die geparste Benutzervorlage

Der Quelltext aus Abbildung 4.15 zeigt die geparste Vorlage. Die Variablen der Vorlage aus dem Quelltext aus Abbildung 4.13 wurden durch den serialisierten Wert des *JSON*-Datenobjekts aus dem Quelltext aus Abbildung 4.14 ersetzt.

Programm 4.15: Das *JSON*-Datenobjekt

```

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 Hugo Maier,</p>
8
9         <p>
10           Hiermit informieren wir Sie &uuml;ber die
11           Status&auml;nderung bez&uuml;glich 
12           <strong>BenutzerIn Status ge&auml;ndert.</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>

```

Die Vorlagenmetadaten

Der folgende Text zeigt die Zeichenkette, welchen die in Abschnitt 3.1.1 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 Die Verwendung in einer Geschäftslogik

Der Quelltext aus Abbildung 4.16 zeigt die implementierte Klasse *EmailServiceCdiEventImpl* der Schnittstelle *EmailService*, die spezifiziert wie über eine Geschäftslogik *E-Mails* erstellt werden können. Die Implementierung *EmailServiceCdiEventImpl* verwendet die implementierte Klasse *CreateEmailsEvent*, die ein *Event*-Objekt darstellt, das über den *CDI-Event-Bus* verarbeitet wird. Es wird ein Objekt der Schnittstelle *javax.enterprise.Event* injiziert, das mit dem *Event*-Objekt typisiert ist. Die Injektionspunkte der *Events* wurden qualifiziert, damit verschiedene Beobachtermethoden aufgerufen werden können.

- *@Immediate*
ist die Annotation, die den injizierten *Event* für die sofortige Ausführung qualifiziert.
- *@AfterSuccess*
ist die Annotation, die den injizierten *Event* für die Ausführung nach dem erfolgreichen Abschluss der Transaktion qualifiziert.

Programm 4.16: Die Klasse *EmailServiceCdiEventImpl*

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

Folgende Auflistung erklärt die implementierten Methoden der Klasse *EmailServiceCdiEventImpl* der Schnittstelle *EmailService*, über die von einer Geschäftslogik aus *E-Mails* erstellt werden können.

- *create(EmailDTO dto)*
ist die Methode, mit der eine *E-Mail* sofort erstellt werden kann, wobei beim Feuern des *Events* die aktuelle Ausführung der Methode unterbrochen und synchron die Beobachtermethode aufgerufen wird.
- *create(List<EmailDTO> dtos)*
ist die Methode, mit der mehrere *E-Mails* sofort erstellt werden können. Das Verhalten der Ausführung ist gleich wie bei der Methode *create(EmailDTO dto)*.

- *createAfterSuccess(EmailDTO dto)*
ist die Methode, mit der eine *E-Mail*, nach dem erfolgreichem Beenden einer Transaktion, erstellt werden kann, wobei die Beobachtermethode trotzdem noch in der Komplettierungsphase der Transaktion aufgerufen wird.
- *createAfterSuccess(List<EmailDTO> dto)*
ist die Methode, mit der mehrere *E-Mails*, nach dem erfolgreichem Beenden einer Transaktion, erstellt werden können. Das Verhalten der Ausführung ist gleich wie in der Methode *createAfterSuccess(EmailDTO dto)*. werden kann.

Der Quelltext aus Abbildung 4.17 zeigt die implementierte 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, welche alle benötigten Informationen für das Erstellen einer *E-Mail* enthält. Das Objekt der Schnittstelle *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. Dieser 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.

Programm 4.17: Die Klasse *BusinessServiceImpl*

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

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 die *Id* des internen *Mail*-Benutzers, welcher die *E-Mail* auf der Datenbank erstellt.
- *mailTypeIds*
ist die Menge von *Ids*, 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 Datumsformatierung definiert.
- *userData*
ist der assoziative Behälter, der die Benutzerdaten enthält, die bei der Ermittlung der aktuellen Werte der Variablen verwendet werden können.

Kapitel 5

Die Tests und erreichten Ziele

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

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

5.1 Die Tests

Dieser Abschnitt beschäftigt sich mit den implementieren Tests des Vorlagenmanagements und der implementierten Konfiguration für die Tests. Für die Tests wurden folgende Bibliotheken verwendet.

- *JUnit4*
ist eine Bibliothek, die ein vollwertiges Test-*Framework* ist, mit dem wiederholbare und reproduzierbare Tests implementiert werden können. *JUnit* ist als Standard für Tests in *Java* anzusehen.
- *DeltaSpike*
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*, bereitstellt.
- *H2*
ist eine Bibliothek, die 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 in einer *CDI*-Umgebung sei auf folgender Blogeintrag von Mark 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, die 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 implementierte Testklasse für die Tests einer Methode der zu testenden Klasse.
- *test_case*
ist der Name der einzelnen Testmethoden, der wiedergibt, was an einer Methode getestet wird.

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

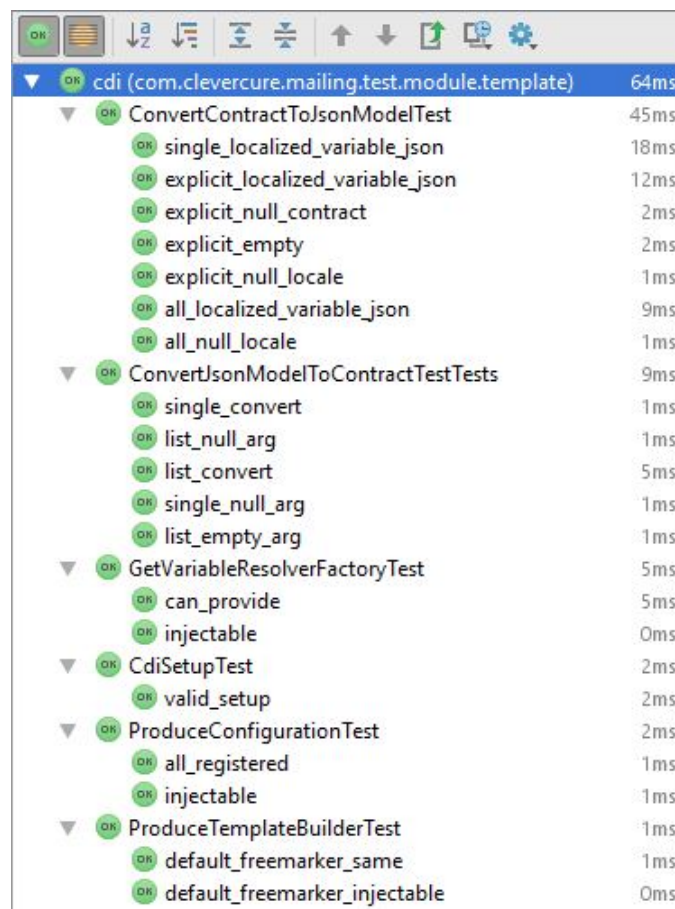
5.1.1 Die Tests der *CDI*-Integration

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

- die Klasse *TemplateCdiExtension*,
- die Klasse *VariableResolverFactoryProvider*,
- die Klasse *CdiTemplateUtils* und
- die Klasse *TemplateResourceProducer* getestet.

Diese Tests sind nur lauffähig in einer *CDI*-Umgebung, die mit *DeltaSpike* 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, müssten Integrationstests implementiert werden, die im verwendeten Anwendungsserver ausgeführt werden. Trotzdem gewährleisten die implementierten Tests, dass die *CDI*-Integration korrekt funktioniert, da die Wahrscheinlichkeit, dass es zu Problemen kommt, äußerst gering ist.



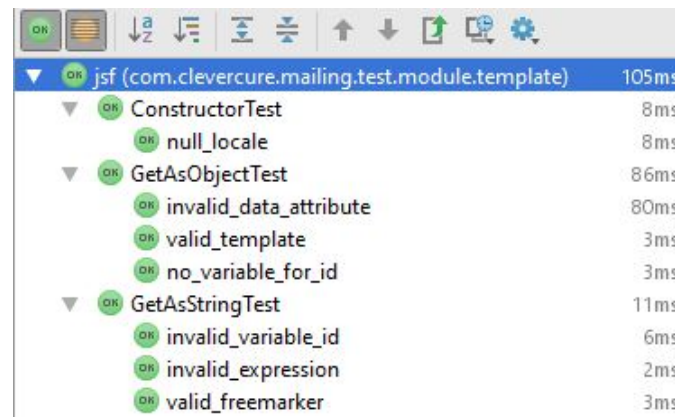
Test Name	Execution Time (ms)
cdi (com.clevercure.mailing.test.module.template)	64ms
ConvertContractToJsonModelTest	45ms
single_localized_variable_json	18ms
explicit_localized_variable_json	12ms
explicit_null_contract	2ms
explicit_empty	2ms
explicit_null_locale	1ms
all_localized_variable_json	9ms
all_null_locale	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 Die 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*-Erweiterung 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 Die 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

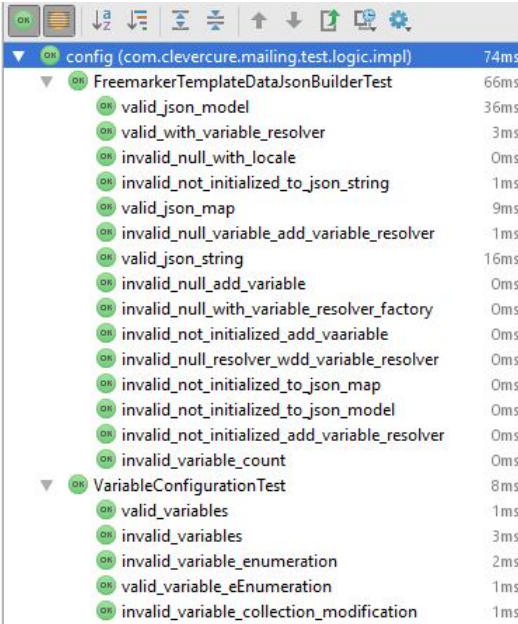
- *VariableConfigurationImpl* und
- *FreemarkerTemplateDataJsonBuilder* getestet.

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 *FreemarkerTemplateDataJsonBuilder* in der Lage ist ein *JSON*-Datenobjekt repräsentiert über ein Objekt der Klasse *TemplateRequestJson* zu produzieren, dass die Daten für eine Voralge hält.

Es sollten noch weitere Tests für die beiden Klassen

- *FreemarkerTemplateProcessor* und
- *FreemarkerTemplateMetadata* implementiert werden.

Die aufgelisteten Klassen werden zwar indirekt über die Tests aus Abbildung 5.3 getestet, sollten trotzdem auch in eigenen Tests getestet werden.



Test Name	Duration
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_vaariable	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 Die erreichten Ziele

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

- *CleverWeb*,
- *CleverSupport* und
- *CleverInterface*.

Die Integration in die Anwendung *CleverInterface* wird warten müssen, 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 sollten sich die Integrationen in die Anwendungen *CleverWeb* und *CleverSupport* einfach und schnell realisieren lassen.

5.2.1 Das *CKEditor-Plugin* für das Vorlagenmanagement

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* implementiert. Das *CKEditor-Plugin* und das Variablenmanagement wurden getrennt voneinander in eigenen Quelltextdateien implementiert. Die implementierten *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. Diese 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 Die *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 eine *Enum* definiert werden. Dazu müsste die Methode *processCdiVariableContracts* der implementierten Klasse *TemplateCdiExtension* und die Klasse *VariableConfigurationImpl* erweitert werden. Die Klasse *TemplateCdiExtension* müsste die registrierten Type 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* verwendet werden müssen.

5.2.3 Das Vorlagenmanagement in *JSF*

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 abstrakte 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 In-

jektionspunkte ersetzen, die auch beim Start der *CDI*-Umgebung validiert werden.

5.2.4 Das Vorlagenmanagement in *Mail-DB*-Schema

Die Integration der Vorlagen in des *Mail-DB*-Schema war die die einfachste Aufgabe, da hier lediglich eine einfache Datenstruktur definiert werden muss, die in der Lage ist, die Vorlagen mehrsprachig persistent zu halten. Prinzipiell ist eine Vorlage auf einer Datenbank als Zeichenkette präsent, wobei nur auf die Größe der Zeichenkette geachtet werden muss. Sollte das Vorlagenmanagement auch in anderen Bereichen verwendet werden, so könnte man eine eigene Datenstruktur definieren, die über *JPA*-Entitäten abgebildet werden könnte und in den verschiedenen Datenbanken verwendet werden könnte.

Kapitel 6

Die Zusammenfassung

Dieses Kapitel beschäftigt sich mit der Zusammenfassung der Bachelorarbeit, der realisierten Implementierung des Vorlagenmanagements und den gemachten Erfahrungen während der Entwicklung. Das implementierte Vorlagenmanagement ist fertiggestellt, wobei sich sicherlich noch neue Anforderungen ergeben werden, 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. Ein Problem könnte die Integration in die Anwendung *CleverInterface* darstellen, da es hier Einschränkungen bezüglich den verwendeten Technologien gibt und man hier sehr stark von der Laufzeitumgebung *IIB* und von der *IBM* abhängig ist.

Es wahr sehr interessant zu sehen, wie leicht sich ein Softwaremodul 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 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 Syntaxfehler 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 viel 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, viel es mir leichter die praktischen Bachelorarbeit auszuarbeiten und die theoretische Bachelorarbeit war eine gute Vorbereitung für die praktische Bachelorarbeit.

Quellenverzeichnis

Literatur

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns*. USA: Addison-Wesley Professional (siehe S. 18).
Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft (2014). *Java 8 In Action*. India: Wiley India (siehe S. 13).

Online-Quellen

Mark Struberg (2012). *Control CDI Containers in SE and EE*. URL: <https://struberg.wordpress.com/2012/03/17/controlling-cdi-containers-in-se-and-ee/> (siehe S. 51).