

Vorlagen-*Management* für *Mail-Service*

ING. THOMAS HERZOG



BACHELORARBEIT

Nr. S1310307011-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Juli 2015

Diese Arbeit entstand im Rahmen des Gegenstands

Gegenstand??

im

Semester??

Betreuer:

FH-Prof. DI Dr. Dobler

Erklärung

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

Hagenberg, am 14. Juli 2015

Ing. Thomas Herzog

Inhaltsverzeichnis

Erklärung	iii
Kurzfassung	vi
Abstract	vii
1 Einleitung	1
1.1 Das Unternehmen curecomp Software Service GmbH	1
1.2 Das Vorlagenmanagement für den <i>Mail</i> -Service	2
1.3 Die Rahmenbedingungen	2
2 Das Ziel des Projekts	4
2.1 Die funktionalen Ziele	5
2.2 Die technischen Ziele	6
3 Das Lösungskonzept	7
3.1 Die Spezifikation der Vorlagen- <i>API</i>	7
3.1.1 Die Schnittstellen und abstrakten Klassen	8
4 Die Realisierung	16
4.1 Die Implementierung der Spezifikationen	16
4.1.1 Die Implementierung für <i>CKEditor</i>	16
4.1.2 Die Implementierungen für CDI	16
4.1.3 Die Implementierungen für JSF	16
4.2 Die Vorlagen- <i>Management</i> Beispielanwendung	16
4.2.1 Die Verwendung in einem <i>Business</i> -Service	16
4.2.2 Die Verwendung in der <i>Web</i> -Oberfläche	16
5 Die Analyse und Tests	17
5.1 Die Tests	17
5.1.1 Die Tests der <i>Services</i>	17
5.1.2 Die Tests der <i>CDI</i> -Integration	17
5.1.3 Die Tests der <i>Web</i> -Oberfläche	17
5.2 Die erreichten Ziele	17

5.2.1	Das Vorlagen- <i>Management über</i> CKEditor	17
5.2.2	Das Vorlagen- <i>Management</i> in einer <i>CDI</i> -Umgebung	17
5.2.3	Das Vorlagen- <i>Management</i> in JSF	17
5.2.4	Das Vorlagen- <i>Management</i> in <i>Mail</i> -DB-Schema	17
A	Technische Informationen	18
A.1	Aktuelle Dateiversionen	18
A.2	Details zur aktuellen Version	18
A.2.1	Allgemeine technische Voraussetzungen	18
A.2.2	Verwendung unter Windows	18
A.2.3	Verwendung unter Mac OS	19
	Quellenverzeichnis	20

Kurzfassung

TODO: Add german summary here

Abstract

TODO: Add english summary here

Kapitel 1

Einleitung

Die vorliegende Sachlage beschäftigt sich mit der Konzeption und Implementierung eines Vorlagen-*Management* für den in der theoretischen Bachelorarbeit konzipierten *Mail-Service*. Das Vorlagen-*Management* stellt einen essentiellen Teil des *Mail-Service* dar, mit dem sich parametrisierte *E-Mail*-Vorlagen erstellen lassen. Das Vorlagen-*Management* soll es den BenutzerInnen ermöglichen einfach eigene parametrisierte *E-Mail*-Vorlagen zu erstellen, die in einer Anwendung, die den *Mail-Service* nutzen, verwendet werden können, um benutzerspezifische *E-Mail*-Nachrichten zu versenden. Mit dem Vorlagen-*Management* 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.

1.1 Das Unternehmen curecomp Software Service GmbH

Diese Arbeit wird in Zusammenarbeit mit dem Unternehmen *curecomp Software Service GmbH* erstellt. Das Unternehmen *curecomp* ist ein Dienstleister im *Supplier-Relationship-Management (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 Schnittstellen-Anwendung für den XML-basierten Datenimport /-export zwischen *clevercure* und den ERP-Systemen der Kunden.
- *CleverSupport* ist eine unternehmensinterne *Web*-Anwendung 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 aller innerhalb *clevercure* anfallender *E-Mail*-Nachrichten.

Wie bereits in der theoretischen Bachelorarbeit behandelt, wird *CCMail* von *CleverMail* abgelöst werden, wobei dass in dieser Arbeit behandelte Vorlagenmanagement die Grundlage für *CleverMail* darstellt. Alle Anwendung innerhalb der Softwarelösung *clevercure* haben die Anforderung das *E-Mail*-Vorlagen parametrisiert und benutzerdefiniert erstellt werden können. Diese Anforderung wird mit dem Vorlagenmanagement erfüllt.

1.2 Das Vorlagenmanagement für den *Mail-Service*

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

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

eliminiert werden. Das Vorlagenmanagement kann auch in einem anderen Kontext verwendet werden, wobei diese Arbeit sich ausschließlich mit der Verwendung des Vorlagenmanagement innerhalb des *Mail-Service* beschäftigen wird.

TODO: Add graphic about template management

1.3 Die Rahmenbedingungen

Das Vorlagenmanagement wird in Java in der Version 8 implementiert und wird sich an der *Java-Enterprise-Edition 7 (JEE-7)* Spezifikation orientieren, wobei folgende Teilspezifikationen Anwendung finden.

- *JPA 2.1* ist die Spezifikation für die Persistenz.
- *CDI 1.1* ist die Spezifikation für kontextabhängige Injektion innerhalb einer *JEE7*-Umgebung.
- *JSF 2.2* ist die Spezifikation der *View*-Technologie.

Damit wird das Vorlagenmanagement mit den aktuellsten Standards imple-

mentiert und wird daher für die Zukunft gut gewappnet sein. Die Funktionalität des Vorlagenmanagement wird weitestgehend ohne die Verwendung spezieller Bibliotheken implementiert, wobei folgende Integrationen zur Verfügung gestellt werden.

- *CDI*-Integration:
Innerhalb eines *CDI-Containers* werden Objekte kontextabhängig zur Verfügung gestellt.
- *JSF*-Integration:
Mit der *View*-Technologie *JSF* wird eine Webseite erstellt, über die die Vorlagen verwaltet werden können.
- *Typescript*-Integration:
Mit *Typescript* wird ein *Plugin* für den *Rich-Editor CKEditor* implementiert, welches die Variablen für eine *E-Mail*-Vorlage innerhalb des *CKEditors* verwaltet.

Als Entwicklungsumgebung wird die *IDE IntelliJ* verwendet, die eine bekannte Entwicklungsumgebung im *Java*-Umfeld darstellt und ein Produkt des Unternehmens *Jetbrains* mit Sitz in Tschechien ist. Als Applikationsserver wird *Wildfly 10.x*, vormals *JbossAS* genannt, des Unternehmens *Redhat* verwendet, der ein zertifizierter *JEE-7*-Server ist und somit alle benötigten Spezifikationen unterstützt.

Kapitel 2

Das Ziel des Projekts

Ziel dieses Projekts ist die Konzipierung und Umsetzung des Vorlagenmanagement für den *Mail-Service*. Das Vorlagenmanagement stellt einen essenziellen Teil des *Mail-Service* dar und wird auch über Anwendungsgrenzen hinweg verwendet werden. Die verschiedenen Anwendungen werden auch unterschiedlichen Laufzeitumgebungen betrieben.

- *IBM-Integration-Bus (IIB)*
ist ein proprietäres Produkt des Unternehmens IBM, für *XML*-Konvertierungen und *XML*-basierten Datenimport und -export.
- *Wildfly*
ist ein zertifizierter *JEE-7* Applikationsserver des Unternehmens *Red-hat*.

Die verschiedenen Anwendungen von *clevercure* sollen mit geringsten Aufwand in der Lage sein *E-Mail*-Vorlagen zu verwenden bzw. *E-Mails* auf Basis dieser Vorlagen zu erstellen und zu versenden. Dabei sollen die benötigten Abhängigkeiten der Anwendungen zu dem Vorlagenmanagement so klein wie möglich gehalten werden, sowie nur vorgegebene Schnittstellen verwendet werden.

Wird eine *E-Mail* aus einer Anwendung heraus auf Basis einer *E-Mail*-Vorlage erstellt, so müssen dessen enthaltene Parameter beim Zeitpunkt des Erstellens aufgelöst und serialisiert werden, damit die *E-Mail* mit denselben Daten zu jedem Zeitpunkt erneut versendet werden kann. Für die Anwendungen soll nicht ersichtlich sein wie die *E-Mails* nach ihrer Erstellung versendet bzw. verwaltet werden. Grundlegend sollen die Anwendungen, die das Vorlagenmanagement verwenden, so geringe Abhängigkeiten wie möglich zum Vorlagenmanagement besitzen.

2.1 Die funktionalen Ziele

Für das Vorlagenmanagement wurden folgende funktionalen Anforderungen definiert.

Die Persistenz der Vorlagen:

Die Vorlagen sollen über Anwendungsgrenzen hinweg innerhalb einer Datenbank persistent gehalten werden. Da das Vorlagenmanagement vorerst exklusiv für den *Mail*-Service verwendet wird, wird die Persistenz der Vorlagen innerhalb des *Mail*-DB-Schema realisiert.

Die Mehrsprachigkeit der Vorlagen:

Die Vorlagen sollen in mehreren Sprachen erstellt und verwaltet werden können, wobei eine Sprache immer als Standardsprache zu definieren ist, auf die zurückgefallen wird, wenn die gewünschte Sprache nicht vorhanden ist.

Die Mehrsprachigkeit der Vorlagenparameter:

Die zur Verfügung stehenden Vorlagenparameter werden durch die EntwicklerInnen spezifiziert und stellen einen Titel und eine Beschreibung des Parameters zur Verfügung. Der Titel und die Beschreibung des Vorlagenparameters sollen mehrsprachig zur Verfügung stehen, wobei als Standardsprache Englisch zu definieren ist.

Die Parametrierbarkeit der Vorlagen:

Die *E-Mail*-Vorlagen werden für einen bestimmten *Mail*-Typ definiert der einen bestimmten Kontext darstellt wie z.B.

- Ein Benutzer wurde erstellt,
- Der Zugang eines Benutzer wurde gesperrt oder
- Das Passwort wurde zurückgesetzt.

Für einen *Mail*-Typ bzw. einen Kontext, in dem eine *E-Mail*-Vorlage verwendet werden, sollen kontextabhängige Vorlagenparameter zur Verfügung stehen wie z.B.:

- *CURRENT_USER* ist der Benutzer der die *E-Mail* erstellt halt.
- *USER* ist der Benutzer, für den die *E-Mail* bestimmt ist.

Die EntwicklerInnen sollen für einen bestimmten Kontext in der Lage sein Parameter zu definieren. Die BenutzerInnen sollen innerhalb der definierten Menge von zur Verfügung stehenden Parametern frei wählen können und eine beliebige Anzahl der Parameter mit Wiederholungen der Parameter in der Vorlage verwenden dürfen.

Die automatische Registrierung der spezifizierten Vorlagenparameter:

Innerhalb einer *CDI*-Umgebung sollen die spezifizierten Vorlagenparameter beim Start des *CDI*-Containers automatisch aufgelöst und registriert werden. Dies ist über eine *CDI-Extension* realisierbar.

Die Verwaltung der Vorlagen über eine Webseite:

Die Vorlagen sollen über eine Webseite verwaltbar sein, wobei eine Vorlage explizit freigegeben werden muss bevor diese Vorlage verwendet wird. Die Webseite soll mit der *View*-Technologie *JSF* implementiert werden. Über einen *FacesConverter* soll die Vorlage von der *View*-Repräsentation in die Repräsentation der verwendeten *Template-Engine* überführt werden.

Da die Vorlagen formatierbar sein sollen wie z.B.: Schriftart und Schriftstil, soll der *Rich-Editor CKEditor* verwendet werden, da es für diesen *JavaScript* basierten *Rich-Editor* bereits eine Integration in *JSF* gibt, die von *Primefaces-Extensions* zur Verfügung gestellt wird.

2.2 Die technischen Ziele

Folgende Punkte wurden als technische Ziele definiert.

Die Verwendung des Vorlagenmanagement in verschiedenen Umgebungen und Anwendungen

Die Kernfunktionalität des Vorlagenmanagement soll ohne Verwendung spezieller Bibliotheken implementiert werden. Diese Funktionalitäten sollen in den verschiedenen Umgebungen wie z.B. einer *CDI*-Umgebung integriert werden können. Innerhalb einer *CDI*-Umgebung können die benötigten Ressourcen vorkonfiguriert und kontextabhängig über Injektion zur Verfügung gestellt werden.

Das Vorlagenmanagement wird in mehreren Anwendungen verwendet, die jeweils unterschiedliche technische Anforderungen definieren, daher soll es jeweils eine eigene Integration pro Anwendung geben, die das Vorlagenmanagement und dessen zur Verfügung gestellten Funktionalitäten in die Anwendung integrieren.

Die Komponentenorientierte Entwicklung

Die einzelnen Softwarekomponenten des Vorlagenmanagement sollen so weit wie möglich unabhängig voneinander implementiert und über Schnittstellen aneinander gekoppelt werden. Damit soll die Modularität des Vorlagenmanagement gewährleistet werden, da es jeweils eigene In

Kapitel 3

Das Lösungskonzept

3.1 Die Spezifikation der Vorlagen-API

Dieses Kapitel behandelt die erstellten Spezifikationen des Vorlagenmanagements. Auf Basis dieser Spezifikationen wird das Vorlagenmanagement und die Integrationen in die verschiedenen Umgebungen realisiert. Die erstellte Spezifikationen sind frei von Abhängigkeiten auf konkrete Implementierungen jeglicher Art. Sie haben nur Abhängigkeiten auf andere Spezifikationen wie die *JEE-7* Spezifikation.

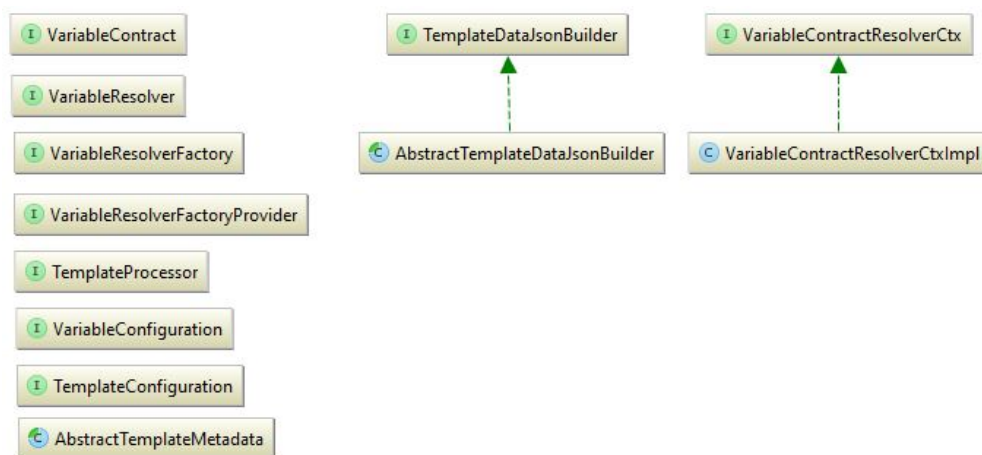


Abbildung 3.1: Klassenhierarchie der Vorlagen-API

3.1.1 Die Schnittstellen und abstrakten Klassen

Dieser Abschnitt behandelt die implementierten Schnittstellen und abstrakten Klassen des Vorlagenmanagements. Die abstrakten Klassen beinhalten die gemeinsam nutzbare Funktionalitäten, welche von allen konkreten Implementierungen des Vorlagenmanagements genutzt werden kann. Diese Spezifikationen spezifizieren folgende Aspekte des Vorlagenmanagements wie

1. das Variablenmanagement innerhalb des Vorlagenmanagement,
2. die Behandlung von Variablen in einer Vorlage
3. die Abbildung der Metadaten einer Vorlage und
4. das Erstellen des *JSON*-Objekts, welches die Daten für die Vorlage beinhaltet.

Die Schnittstelle *VariableContract*

Die Schnittstelle *VariableContract* spezifiziert eine Variable, die in einer Vorlage verwendet werden kann. Objekte dieser Schnittstelle werden einmalig registriert und können in allen Vorlagen verwendet werden. Eine Variable ist einem Modul zugeordnet, in dem die Variable bezüglich ihres Namen eindeutig sein muss. Das Modul wird über einen *String* definiert. Die Mehrsprachigkeit eines Variablenkontrakt wird über Enumerationen realisiert, wobei ein jeder Variablenkontrakt jeweils einen Schlüssel für den *Label* und die Beschreibung bereit stellt.

Da es sich bei einer Variablenkontrakt um statische Daten handelt, also die Variablen schon zur Kompilierungszeit bekannt sind, ist angedacht, dass die Variablen mit dem *Java*-Typ *enum* definiert werden, wobei in einer Klasse von Typ *enum* mehrere Variablen definiert auf einmal werden können. Diese definierten Variablen innerhalb dieser Klasse sollten demselben Modul zugeordnet sein. 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.

Programm 3.1: VariableContract.java

```
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 }
```

Ein Variablenkontrakt ist über seine Id eindeutig referenzierbar, wobei sich die Id aus dem Modulnamen und den Variablennamen zusammensetzt (Bsp. `module.core.VARIBALE`). Dieses Verhalten soll immer gleich sein, deswegen wurde die Methode *String getId()*; als *default Methode* implementiert, was seit *Java8* möglich ist. Ein EntwicklerIn muss diese Methode nicht mehr implementieren, obwohl es immer noch möglich ist diese Methode zu überschreiben. Auch die Methode *String toInfoString()* wurde als *default Methode* implementiert, da auch diese Methode nicht von den EntwicklerInnen implementiert werden sollte, da ihre Funktionalität nicht ändern sollte.

Die Schnittstelle *VariableResolver*

Die Schnittstelle *VariableResolver* spezifiziert wie Variablen aufgelöst werden. Eine Variable wird in einer Vorlage verwendet und beim Parsen dieser Vorlage muss der aktuelle Wert der Variable aufgelöst werden. Da dieser Variablenwert abhängig ist vom Kontext der Vorlage wird beim Auflösen der Variable ein Kontext bereitgestellt, über den kontextabhängige Daten vom EntwicklerIn bereitgestellt werden können, die in einer Implementierung der Schnittstelle *VariableResolver* angewendet werden können. Dadurch könnte die Variable in mehreren Kontexten verwendet werden und auch unterschiedlich aufgelöst werden abhängig vom aktuell gesetzten Kontext.

Programm 3.2: VariableResolver.java

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

Die Schnittstelle wurde als *FunctionalInterface* implementiert. Ein *FunctionalInterface* ist eine Schnittstelle, die nur eine abstrakte Methode definiert, die implementiert werden muss. Eine Implementierung eines *FunctionalInterface* kann über eine *Lambda*-Funktion oder Methodenreferenzen bereitgestellt werden, wodurch die Notwendigkeit einer anonymen Implementierung oder der Implementierung einer Klasse wegfällt. Dieser Ansatz 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 begründet ist. Die negativen Auswirkungen auf das Laufzeitverhalten können im Bezug auf das Vorlagenmanagement vernachlässigt werden.

Die Schnittstelle *VariableResolverFactory*

Die Schnittstelle *VariableResolverFactory* spezifiziert wie Objekte der Schnittstelle *VariableResolver* produziert werden. Objekte dieser Schnittstelle können Implementierungen der Schnittstelle *variableResolver* für jede Art von Variablen produzieren, obwohl es zu empfohlen ist, dass es eine Implementierung der Schnittstelle *VariableResolverFactory* je Variablenmodul gibt.

Programm 3.3: VariableResolverFactory.java

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

Auch diese Schnittstelle wurde als *FunctionalInterface* implementiert, damit Implementierungen über *Lambda*-Funktionen oder Methodenreferenzen bereitgestellt werden können.

Die Schnittstelle *VariableResolverFactoryProvider*

Die Schnittstelle *VariableContractFactoryProvider* spezifiziert wie Objekte der Schnittstelle *VariableResolverFactory* produziert werden. Ein Objekt der Schnittstelle *VariableResolverFactoryProvider* kann Objekte der Schnittstelle *VariableResolverFactory* für eine konkrete Implementierung der Schnittstelle *VariableContract* zur Verfügung stellen. Es ist angedacht, dass die Variablen mit dem *Java*-Typ *enum* definiert werden, wobei die Implementierung eines *Enum*-Typs die Schnittstelle *VariableContract* implementiert. Dadurch definiert jede einzelne Enumeration einen Variablenkontrakt.

Programm 3.4: VariableResolverFactoryProvider.java

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

Auch diese Schnittstelle wurde als *FunctionalInterface* implementiert um Implementierungen über *Lambda*-Funktionen oder Methodenreferenzen zur Verfügung stellen zu können.

Die Schnittstelle *VariableContractResolverCtx*

Die Schnittstelle *VariableContractResolverCtx* spezifiziert den Kontext, welcher bei der Auflösung der Variablen zur Verfügung steht. Dieser Kontext stellt alle Daten bereit, die bei der Auflösung der Variable für eine Vorlage nötig sind. Es ist auch möglich Benutzerdaten im Kontext zu setzen, die

bei der Auflösung der Variablen angewendet werden können, wodurch das Auflösen der Variablen unabhängig von der Handhabung der Vorlage ist.

Programm 3.5: VariableContractResolverCtx.java

```
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* spezifiziert wie die Vorlagen behandelt werden. Objekte dieser Schnittstelle können Variablen in einer Vorlage, einer bestimmten *Template-Engine* finden und konvertieren. Ein *TemplateProcessor* muss ebenfalls in der Lage sein ungültige Variablen innerhalb einer Vorlage zu finden, wobei eine ungültige Variable eine Variable ist, die nicht innerhalb des aktuellen Kontextes nicht gefunden werden kann und somit auch nicht aufgelöst werden kann.

Eine konkrete Implementierung dieser Schnittstelle ist eine Implementierung für eine bestimmte *Template-Engine*, da die in der Vorlage verwendeten *Expressions* spezifisch für die verwendete *Template-Engine* sind.

Die Methoden *String replaceExpression(...)* und *String replaceCustom(...)* verwenden als Formalparameter für den benötigte Konverter ein *Functional-Interface* namens *Function*, welches von der Sprache *Java* 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.

Programm 3.6: TemplateProcessor.java

```
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* spezifiziert die Signatur des konkreten *Builders*, der das *JSON*-Objekt erstellt, welches die Daten für das Parsen einer Vorlage enthält. Eine Anforderung ist es die *E-Mail*-Nachrichten persistent zu halten, wobei der Inhalt der *E-Mail*-Nachricht unveränderbar sein soll. Daher wurde entschieden dass die Daten einer *E-Mail*-Nachricht in Form eines *JSON*-String persistent gehalten werden. Mit diesem *JSON*-Objekt kann die korrespondierende Vorlage zu jedem Zeitpunkt mit demselben Resultat erneut geparkt werden.

Es wurde hier das *Builder*-Muster angewendet, da sich die Initialisierung des *Builders* mit einer *Fluent-API*, wie bei einem *Builder* üblich, sehr gut abbilden lässt. Folgendes Beispiel soll illustrieren, wie ein *Builder* initialisiert werden kann.

```
builder.withLocalization(locale, zoneId)
        .withTemplate(templateString)
        .withUserData(userDataMap)
        .toJsonModel();
```

Programm 3.7: TemplateDataJsonBuilder.java

```
1 public interface TemplateDataJsonBuilder<I,  
2     M extends AbstractTemplateMetadata<I>,  
3     B extends TemplateDataJsonBuilder>  
4     extends Serializable {  
5  
6     B withWeakMode();  
7  
8     B withLocalization(Locale locale,  
9         ZoneId zoneId);  
10  
11     B withUserData(Map<Object, Object> userData);  
12  
13     B withStrictMode();  
14  
15     B withVariableResolverFactoryFactory(VariableResolverFactoryProvider  
16         factory);  
17  
18     B withVariableResolverFactory(VariableResolverFactory factory);  
19  
20     B withTemplate(M metadata);  
21  
22     void end();  
23  
24     B addVariable(VariableContract contract,  
25         Object value);  
26  
27     B addVariableResolver(VariableContract contract,  
28         VariableResolver resolver);  
29  
30     TemplateRequestJson toJsonModel();  
31  
32     String toJsonString();  
33  
34     Map<String, Object> toJsonMap();  
35 }
```

Die abstrakte Klasse *AbstractTemplateMetadata*

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

1. die Anzahl der gültigen Variablen in der Vorlage,
2. die Anzahl der ungültigen Variablen in der Vorlage,
3. die Zeichenlänge der Vorlage,
4. die eindeutige *Id* der Vorlage,
5. die Version der Vorlage und

6. die Vorlage selbst

werden in dieser Klasse abgebildet. Diese Metadaten sind unabhängig der verwendeten *Template-Engine* und eine konkrete Implementierung für eine *Template-Engine* kann zusätzliche Metadaten definieren. Die Metadaten werden einmalig ermittelt und sind über die Lebenszeit des Objekts unveränderbar.

TODO: Add reference to appendix for this source

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder*

Die abstrakte Klasse *AbstractTemplateDataJsonBuilder* implementiert die Logik, die von allen konkreten Implementierungen genutzt werden kann. Sie stellt ebenso Hilfsmethoden bereit, die Variablen innerhalb der Vorlage finden, validieren und auflösen. Das resultierende *JSON*-Objekt des *Builders* ist spezifiziert, jedoch nicht die Struktur der Daten für die beinhalteten Variablen. Diese Daten sind spezifisch für die verwendete *Template-Engine*. Es könnten auch noch andere Daten für das Verarbeiten einer Vorlage von Nöten sein, die in der spezifizierten *JSON*-Objekt nicht vorhanden sind.

TODO: Add reference to appendix for this source

Kapitel 4

Die Realisierung

4.1 Die Implementierung der Spezifikationen

4.1.1 Die Implementierung für *CKEditor*

Das *CKEditor-Plugin* in Typescript

Die Variablenrepräsentation in JSON

4.1.2 Die Implementierungen für CDI

Die Vorlagen-*Management CDI-Extension*

Der Vorlagen-*Management CDI-Producer*

Die Vorlagen-*Management CDI-Utility*

4.1.3 Die Implementierungen für JSF

Der Vorlagen *FacesConverter*

Die *Primefaces-Extension* für den *CKEditor*

4.2 Die Vorlagen-*Management* Beispielanwendung

4.2.1 Die Verwendung in einem *Business-Service*

4.2.2 Die Verwendung in der *Web-Oberfläche*

Kapitel 5

Die Analyse und Tests

5.1 Die Tests

5.1.1 Die Tests der *Services*

5.1.2 Die Tests der *CDI*-Integration

5.1.3 Die Tests der *Web*-Oberfläche

5.2 Die erreichten Ziele

5.2.1 Das Vorlagen-*Management* über CKEditor

5.2.2 Das Vorlagen-*Management* in einer *CDI*-Umgebung

5.2.3 Das Vorlagen-*Management* in JSF

5.2.4 Das Vorlagen-*Management* in *Mail*-DB-Schema

Anhang A

Technische Informationen

A.1 Aktuelle Dateiversionen

Datum	Datei
2015/09/19	hgbthesis.cls
2015/11/04	hgb.sty

A.2 Details zur aktuellen Version

Das ist eine völlig überarbeitete Version der DA/BA-Vorlage, die UTF-8 kodierten Dateien vorsieht und ausschließlich im PDF-Modus arbeitet. Der „klassische“ DVI-PS-PDF-Modus wird somit nicht mehr unterstützt!

A.2.1 Allgemeine technische Voraussetzungen

Eine aktuelle LaTeX-Installation mit

- Texteditor für UTF-8 kodierte (Unicode) Dateien,
- **biber**-Programm (BibTeX-Ersatz, Version ≥ 1.5),
- **biblatex**-Paket (Version ≥ 2.5 , 2013/01/10),
- Latin Modern Schriften (Paket **lmodern**).¹

A.2.2 Verwendung unter Windows

Eine typische Installation unter Windows sieht folgendermaßen aus (s. auch Abschnitt ??):

1. **MikTeX 2.9**² (zurzeit am einfachsten die 32-Bit Version, da nur diese

¹<http://www.ctan.org/pkg/lm>, <http://www.tug.dk/FontCatalogue/lmodern>

²<http://www.miktex.org/> – **Achtung:** Generell wird die **Komplettinstallation** von MikTeX („Complete MiKTeX“) empfohlen, da diese bereits alle notwendigen Zusatzpakete und Schriftdateien enthält! Bei der Installation ist darauf zu achten, dass die automatische

- das Programm `biber.exe` bereits enthält),
- 2. **TeXnicCenter 2.0**³ (Editor-Umgebung, unterstützt UTF-8),
- 3. **SumatraPDF**⁴ (PDF-Viewer),

Ein passendes TeXnicCenter-Profil für MikTeX, Biber und Sumatra ist in diesem Paket enthalten (Datei `_tc_output_profile_sumatra_utf8.tco`). Dieses sollte man zuerst über **Build** → **Define Output Profiles** in TeXnicCenter importieren. **Achtung:** Alle neu angelegten `.tex`-Dateien sollten in UTF-8 Kodierung gespeichert werden!

A.2.3 Verwendung unter Mac OS

Diese Version sollte insbesondere mit *MacTeX* problemlos laufen (s. auch Abschnitt ??):

1. *MacTeX* (2012 oder höher).
2. Die Zeichenkodierung des Editors sollte auf UTF-8 eingestellt sein.
3. Als Engine (vergleichbar mit den Ausgabeprofilen in TeXnicCenter) sollte *LaTeXmk* verwendet werden. Dieses Perl-Skript erkennt automatisch, wie viele Aufrufe von *pdfLaTeX* und *Biber* nötig sind. Die Ausgabeprofile *LaTeX* oder *pdfLaTeX* hingegen müssen mehrmals aufgerufen werden, zudem werden hierbei auch die Literaturdaten nicht verarbeitet. Dazu müsste extra die *Biber*-Engine aufgerufen werden, die jedoch noch nicht in allen Editoren vorhanden ist.

Installation erforderlicher Packages durch „*Install missing packages on-the-fly*: = *Yes*“ ermöglicht wird (NICHT „*Ask me first*“)! Außerdem ist zu empfehlen, unmittelbar nach der Installation von MikTeX mit dem Programm **MikTeX** → **Maintenance** → **Update** und **Package Manager** ein Update der installierten Pakete durchzuführen.

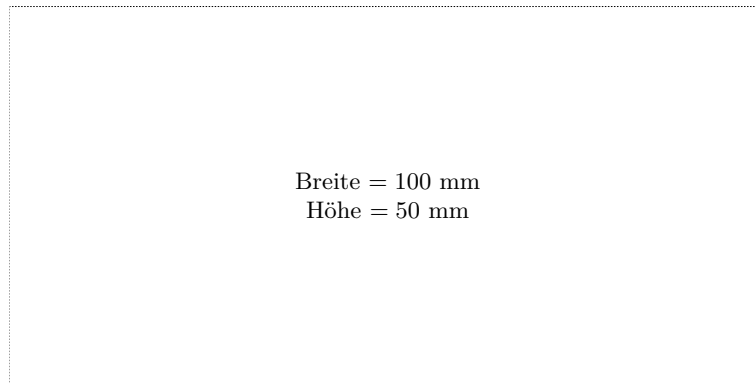
³<http://www.texniccenter.org/>

⁴<http://blog.kowalczyk.info/software/sumatrapdf/>

Quellenverzeichnis

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —