Vorlagenmanagement 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

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 Vorlagenmanagement für den, in der theoretischen Bachelorarbeit konzipierten, *Mail*-Service. Das Vorlagenmanagement stellt einen essentiellen Teil des *Mail*-Service 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 einer Anwendung, die den *Mail*-Service 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 wurde entschieden sich vorerst auf das Vorlagenmanagement zu konzentrieren. Das Vorlagenmanagement wird für den *Mail*-Service 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 aufweisen.

1.1 Das Unternehmen curecomp Software Service GmbH

Das Vorlagenmanagement wird in Zusammenarbeit mit dem Unternehmen *curecomp Software Service GembH* erstellt. Das Unternehmen *curecomp* ist ein 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 Schnittstellenanwendung für den XML-basierten Datenimport und Datenexport zwischen *clevercure* und den *ERP*-Systemen der Kunden.
- *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-Mail*-Nachrichten innerhalb von *clevercure*, die durch *CleverMail* abgelöst werden soll.

Das Vorlagenmanagement wird von den Anwendungen innerhalb von *clevercure* verwendet werden bevor der *Mail*-Service fertiggestellt wird, da es bereits Softwarekomponenten innerhalb der Anwendungen von *clevercure* gibt, die darauf angewiesen sind.

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 sich die vorliegende Sachlage ausschließlich mit der Verwendung des Vorlagenmanagement innerhalb des *Mail*-Service beschäftigen wird. Obwohl das Vorlagenmanagement als eigene Softwarekomponente implementiert wird, wird die vorliegende Sachlage aufzeigen, wie sich das Vorlagenmanagement in Anwendungen im Kontext von *E-Mail*-Vorlagen verwendet lässt.

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 werden:

- *Java-Persistence-API 2.1 (JPA)* ist die Spezifikation für die Persistenz.
- *Context and dependency Injection 1.1 (CDI)* ist die Spezifikation für kontextabhängige Injektion innerhalb einer *JEE7*-Umgebung.
- *Java-Server-Faces 2.2 (JSF)* ist die Spezifikation der *View*-Technologie.

Damit wird das Vorlagenmanagement mit den aktuellsten Standards implementiert und wird daher für die Zukunft gut gewappnet sein. Die Funktionalität des Vorlagenmanagement wird weitestgehend ohne die Verwendung spezieller Bibliotheken implementiert. Es werden Integrationen des Vorlagenmanagement in die folgende Technologien implementiert:

- Integration in *CDI*:
  Innerhalb einer *CDI*-Umgebung werden Ressourcen des Vorlagenmanagements kontextabhängig zur Verfügung gestellt.
- Integration in *JSF*:
  Mit der *View*-Technologie *JSF* wird eine Webseite erstellt, über welche die Vorlagen verwaltet werden.
- Integration in *Typescript*:
  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 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 wird so weit wie möglich vermieden 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

Ziel ist es die Softwarekomponente Vorlagenmanagement für den *Mail*-Service zu implementieren, mit dem *E-Mail*-Vorlagen erstellt und verwaltet werden können. Das Vorlagenmanagement stellt einen essentiellen Teil des *Mail*-Service dar und wird auch von mehreren Anwendungen verwendet werden. Die verschiedenen Anwendungen, die das Vorlagenmanagement verwenden, sind ebenfalls in Java implementiert, werden aber in unterschiedlichen Laufzeitumgebungen betrieben wie z.B.:

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

Die verschiedenen Anwendungen von *clevercure* sollen mit geringsten Aufwand in der Lage sein *E-Mail*-Vorlagen zu verwenden und *E-Mail*-Nachrichten auf Basis dieser *E-Mail*-Vorlagen zu erstellen. Dabei sollen die Abhängigkeiten der Anwendungen zu dem Vorlagenmanagement so gering wie möglich gehalten werden, sowie nur vorgegebene Schnittstellen verwendet werden. Wird eine *E-Mail*-Nachricht von einer Anwendung auf Basis einer *E-Mail*-Vorlage erstellt, so müssen dessen enthaltene Variablen beim Zeitpunkt des Erstellens der *E-Mail*-Nachricht aufgelöst und serialisiert werden, damit die *E-Mail*-Nachricht mit denselben Daten zu jedem Zeitpunkt erneut versendet werden kann. Für die Anwendungen soll 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 dessen Datenbankschema.

2.1 Die funktionalen Ziele

Für das Vorlagenmanagement wurden die folgende funktionalen Anforderungen definiert.

2.1.1 Die Persistenz der Vorlagen

Die Vorlagen müssen innerhalb einer Datenbank persistent gehalten werden. Da das Vorlagenmanagement vorerst exklusiv für den *Mail*-Service verwendet wird, soll die Persistenz der Vorlagen innerhalb des *Mail*-DB-Schema realisiert werden. Die persistenten Vorlagen müssen versioniert werden, damit diese von anderen Entitäten referenziert werden können, ohne dass die Gefahr besteht, dass sich die referenzierte Vorlage geändert hat, wodurch die Konsistenz verloren gehen würde. Persistente Vorlagen müssen explizit freigegeben werden bevor diese verwendet dürfen. Nach einer Freigabe darf die Vorlage nicht mehr geändert werden.

2.1.2 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 immer eine Vorlage für jede angeforderte Sprache zur Verfügung steht. Es ist nicht erforderlich dass dieselben Variablen über alle definierte Sprachen gleich sind. Es dürfen in einer Vorlage, die in mehreren Sprachen definiert wurde, eine unterschiedliche Anzahl oder unterschiedliche Variablen definiert sein.

2.1.3 Die Variablen für die Vorlagen

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

- ein Benutzer wurde erstellt,
- eine Bestellung wurde erstellt oder
- ein Dokument wurde hochgeladen.

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

- *CURRENT_USER* ist der Benutzer, der die *E-Mail*-Nachricht erstellt halt.
- *ORDER_NUMBER* ist die Nummer der erstellten Bestellung.

Die EntwicklerInnen sollen 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 sollen auch global definiert werden können und in allen Vorlagen anwendbar 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.

2.1.4 Die Mehrsprachigkeit der Variablen

Die zur Verfügung stehenden Variablen werden durch die EntwicklerInnen statisch definiert und müssen einen Titel und eine Beschreibung einer Variable zur Verfügung stellen. Der Titel und die Beschreibung der Variable müssen mehrsprachig zur Verfügung stehen, wobei als Standardsprache Englisch zu verwenden ist.

2.1.5 Die automatische Registrierung der Variablen

Innerhalb einer *CDI*-Umgebung sollen die definierten Variablen beim Start des *CDI-Containers* automatisch gefunden und registriert werden. Die automatische Registrierung der Variablen soll mit einer *CDI-Extension (javax.inject.spi.Extension)* realisiert werden, die beim Start des *CDI-Containers*, die Variablen findet und registriert. Mit einer automatischen Registrierung der variablen wird erreicht das neu definierte Variablen automatisch gefunden und registriert werden und somit nicht manuell registriert werden müssen, was ein gewisses Risiko in sich birgt, wenn Variablen vergessen werden.

2.1.6 Die Verwaltung der Vorlagen über eine Webseite

Die Vorlagen sollen über eine Webseite verwaltet werden können. 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* konvertiert werden.
Das folgende *HTML-Markup* enthält die Variablen in ihrer *HTML*-Repräsentation wie sie in dem *Rich-Editor* verwendet wird.

Programm 2.1: *HTML-Markup* einer Vorlage

```
<p>Das ist eine Variable:</p>
<span class="variable" title="Beschreibung" data-variable="VAR_1">
  Variable 1
</span>
```

Der folgende Text stellt das konvertierte *HTML-Markup* aus 2.2 als *Freemarker-Template* dar.

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

```
<p>Das ist eine Variable:</p>
${module.core.VariableHolder["VAR_1"]!("Variable nicht gefunden")}
```

Als *Rich-Editor* soll *CKEditor* verwendet werden, da es für diesen *Rich-Editor* von *primefaces-extensions* eine *JSF*-Integartion in Form einer *JSF*-Komponente zur Verfügung gestellt wird. Dadurch entfällt die Integration eines reinen *Javascript Rich-Editors* der keine Integartion in den Lebenszyklus von *JSF* hat und daher auch keine *AJAX*-Events unterstützt, die von *JSF* verarbeitet werden können.

2.2 Die technischen Ziele

Als technische Ziele wurde die Implementierung in *Java 8*, die Integration in eine *CDI*-Umgebung und die komponentenorientierte Entwicklungdes Vorlagenmanagements definiert. Das Templatemanagement soll als eine eingeständige Softwarekomponenete implementiert werden, die ohne großen Aufwand in anderen Anwendungen verwendet werden kann, sofern die technischen Vorraussetzunge wie die Version von *Java* und die Unterstützung der verwendeten Bibliotheken, erfüllt sind. Das Vorlagenmanagement soll Schnittstellen definieren, die Funktionalitäten des Vorlagenmanagements nach außen offenlegen, ohne dass die Anwendungen in Berüghrung mit konkreten Implementierungen kommen.

Kapitel 3

Das Lösungskonzept

In diesem Kapitel wird der Lösungsansatz und die Spezifikation des Vorlagenmanagements erörtert. Bei der Spezifikation handelt es sich um Schnittstellen und abstrakte Klassen, die die Struktur des Vorlagenmanagements definieren. Diese Schnittstellen und abstrakten erlauben es Implementierungen für verschiedene *Template-Engines* wie z.B

- *Freemakrer*,
- *Velocity* oder
- *Thymeleaf*

zur Verfügung zu stellen, wobei die abstrakten Klassen die gemeinsam nutzbare Logik implementieren, die über die verschiedenen *Template-Engines* verwendet werden kann.

Mit der Möglichkeit die verschiedensten *Template-Engines* verwenden zu können, wird erreicht dass das Vorlagenmanagement sehr flexibel ist. Bei dem Wechsel zu einer anderen *Template-Engine* müssen nur die *Expressions* einer Vorlagen in die *Template-Engine* spezifischen *Expressions* umgewandelt werden.

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.

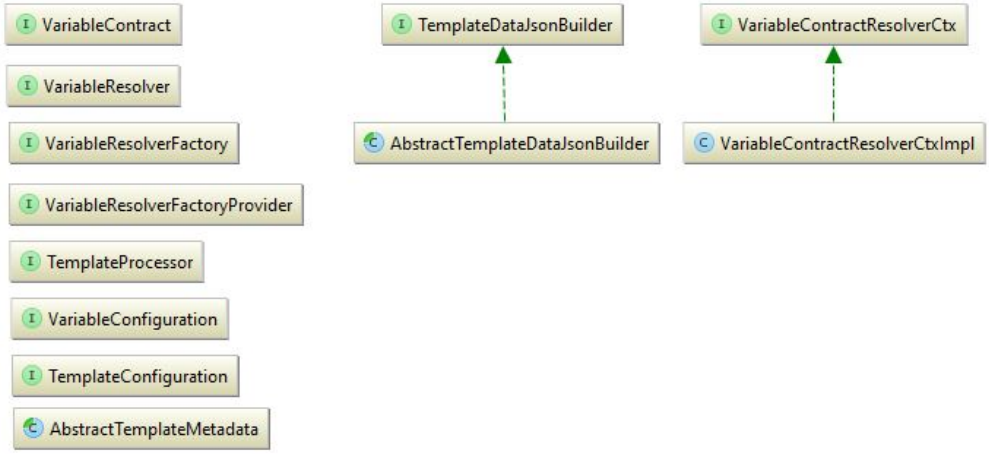


Abbildung 3.1: Klassenhierarchie der Vorlagen-*API*

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 alle 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 Voralge 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 sind 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 *VariableResolverFacotry* 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 setzten, 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 *FunctionalInterface* 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>
      converter);
5
6     String replaceCustom(String template,
7                          Pattern itemPattern,
8                          Function<String, String> converter);
9
10    Set<VariableContract> resolveExpressions(String template);
11
12    Set<String> resolveInvalidExpressions(String template);
13
14    String variableToExpression(VariableContract contract);
15
16    VariableContract expressionToVariable(String expression);
17 }
```

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 geparst 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
       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 }
```

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 $\geq$ 1.5),
- `biblatex`-Paket (Version $\geq$ 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 enhalten (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.

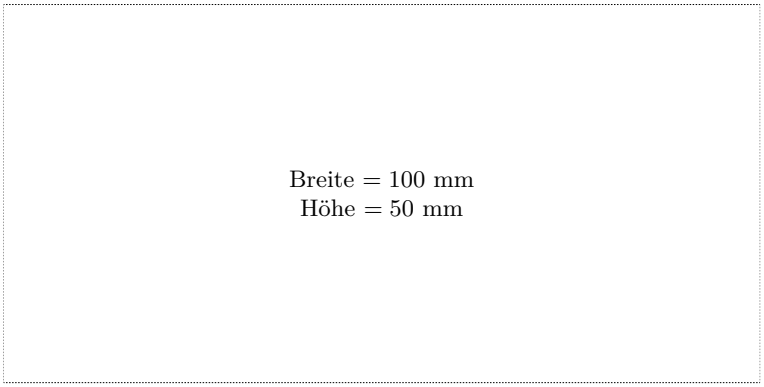
[3] http://www.texniccenter.org/

[4] http://blog.kowalczyk.info/software/sumatrapdf/

Quellenverzeichnis

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —