Automatic Value Propagation in Code Generator Templates

More Examples on the Automatic Value Propagation Mechanism in the Template Engine of the JIOWA Code Generation Framework

Dr. Robert Mencl
Freelance IT Consultant / www.mencl.de

JIOWA Business Solutions GmbH
Bettinastraße 30
D-60325 Frankfurt am Main
Germany

codegen@jiowa.de
www.jiowa.de

Version 2.1.6, May 3rd, 2017
www.jiowa.de/download.html

Automatic Value Propagation in Code Generator Templates

1. Email Template
2. Multilingual Templates
3. Java Class Template
4. Java Class with Inline Templates
5. Summary

Email Template

1. Email Template

Template file: `Letter.jgt`

```
Dear <<Salutation --> Mr: {Mr.} | Mrs: {Mrs.}>> <<Name>>,

we hereby want to inform you...
bla bla bla .

Best regards,

<<ContactPerson>>
```

Plain text enriched with template notation elements which are enclosed by << ... >>

Automatic value propagation which is explained in this presentation, is a feature of the template engine of the JIOWA Code Generation Framework!

If you are not familiar with the framework you should check the tutorial here:

Slides and quick introduction: www.jiowa.de/products/#jiowa-codegen

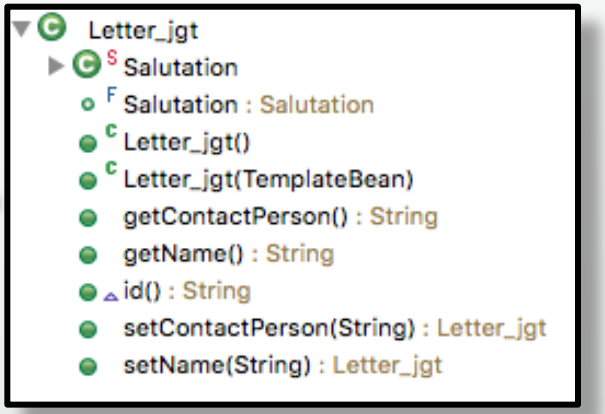
1. Email Template (2)

Template file: `Letter.jgt`

```
Dear <<Salutation -->
       Mr: {Mr.} | Mrs: {Mrs.}>> <<Name>>,

we hereby want to inform you...
bla bla bla .

Best regards,

<<ContactPerson>>
```

Automatic Build

TemplateBean Class: Letter_jgt



Using the template bean in your program:

```
Letter_jgt template = new Letter_jgt();

template.Salutation.set_Mr();
template.setName("Smith")
        .setContactPerson("Jenny Jones");

System.out.println(template);
```

Output

```
Dear Mr. Smith,

we hereby want to inform you...
bla bla bla .

Best regards,

Jenny Jones
```

How can we write email templates for different languages but insert the data only once? ➠ The solution is on the next slide!

Multilingual Templates

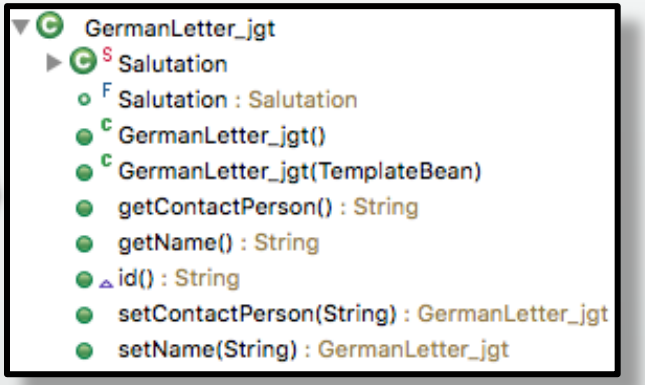
2. Multilingual Templates

What if we have multiple templates with the same logical structure but just different visual appearence?

Template file: `GermanLetter.jgt`

```
Sehr <<Salutation --> Mr:  {geehrter Herr} |
                      Mrs: {geehrte Frau}>> <<Name>>,

wir möchten Sie hiermit darauf hinweisen...
bla bla bla ...

Mit freundlichen Grüßen

<<ContactPerson>>
```

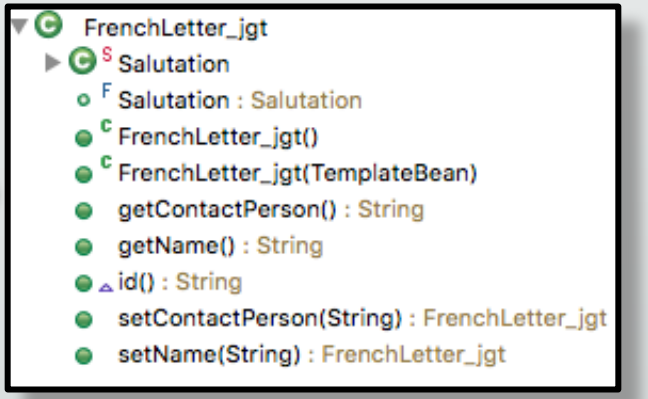
Automatic Build



Template file: `FrenchLetter.jgt`

```
<<Salutation --> Mr: {Cher Monsieur} |
                 Mrs: {Chère Madame}>> <<Name>>,

nous aimerions vous signaler...
bla bla bla...

Avec nous meilleurs sentiments,

<<ContactPerson>>
```

Automatic Build



2. Multilingual Templates (2)

What if we have multiple templates with the same logical structure but just different visual appearence?

Using letter templates in different languages at the same time:

```
Letter_jgt letter = new Letter_jgt();
letter.Salutation.set_Mr();
letter.setName("Bean").setContactPerson("Mary Moneypenny");

// two examples for automatic value propagation
// via 'parent constructor' initialization:
GermanLetter_jgt brief  = new GermanLetter_jgt(letter);
FrenchLetter_jgt lettre = new FrenchLetter_jgt(letter);

System.out.println(letter);
System.out.println(brief);
System.out.println(lettre);
```

```
Sehr geehrter Herr Bean,

wir möchten Sie hiermit darauf
hinweisen...
bla bla bla .

Mit freundlichen Grüßen

Mary Moneypenny
```

```
Cher Monsieur Bean,

nous aimerions vous signaler...
bla bla bla .

Avec nous meilleurs sentiments,

Mary Moneypenny
```

This is called automatic value propagation!

2. Multilingual Templates (3)

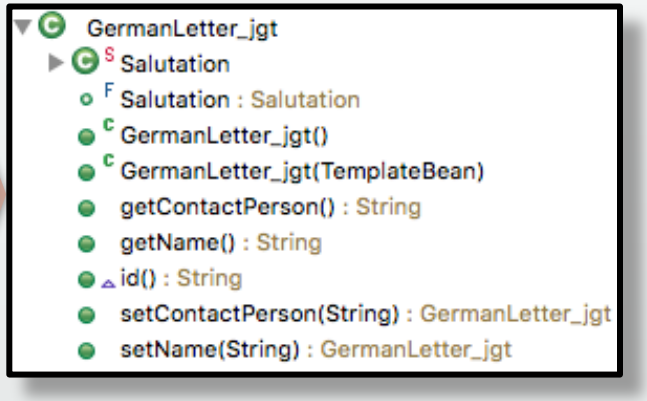
Why does this value propagation work?

Same names for inline subtemplates and variables!

Template file: `GermanLetter.jgt`

```
Sehr <<Salutation --> Mr:  {geehrter Herr} |
                      Mrs: {geehrte Frau}>> <<Name>>,

wir möchten Sie hiermit darauf hinweisen...
bla bla bla ...

Mit freundlichen Grüßen

<<ContactPerson>>
```

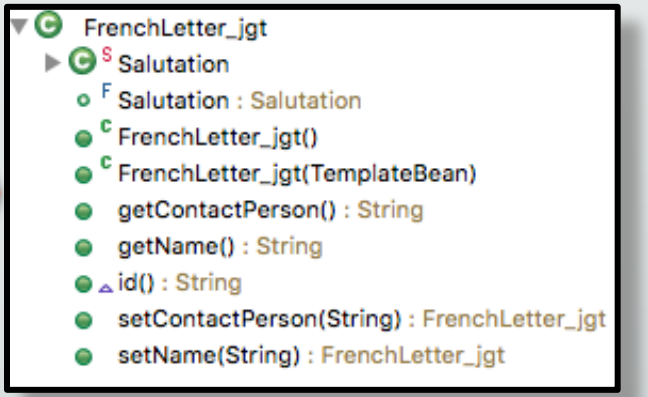
Automatic Build



Template file: `FrenchLetter.jgt`

```
<<Salutation --> Mr: {Cher Monsieur} |
                 Mrs: {Chère Madame}>> <<Name>>,

nous aimerions vous signaler...
bla bla bla...

Avec nous meilleurs sentiments,

<<ContactPerson>>
```

Automatic Build



2. Multilingual Templates (4)

Properties of the value propagation mechanism:

- If the value of a variable (like `<<Name>>` or `<<ContactPerson>>`) in the child template bean is null, the value of the parent template bean is taken.
- If the value of a variable in a template bean is never initialized, an error is logged during code generation.
- If the list of sub template beans (like `Mr`, `Mrs`) of a sub structure (like `Salutation`) is null, the list of the parent template bean is used instead.
- Unlike variables, sub structures are optional elements.
  If no sub template beans have been set or added, no text will be created for this sub structure.

Java Class Template

3. Java Class Template

Template file: `Class.jgt`

```
package <<PackageName>>;

public class <<ClassName>>
{
    << foreachAttribute --> Attribute.jgt />>

    << Include <-- Constructor.jgt />>

    << foreachAttribute --> Getter.jgt />>

    // {{ProtectedRegionStart::ManuallyWrittenCode}}
        // ...
        // insert your customized code here!
        // ...
    // {{ProtectedRegionEnd}}
}
```

Java Class Template with Includes and Sub Templates

Includes & sub templates can be used to significantly reduce the visual complexity of templates!

3. Java Class Template (2)

Java Class Template with Includes and Sub Templates

Template file: `Class.jgt`

```
package <<PackageName>>;

public class <<ClassName>>
{
    << foreachAttribute --> Attribute.jgt />>

    << Include <-- Constructor.jgt />>

    << foreachAttribute --> Getter.jgt />>

    // {{ProtectedRegionStart::ManuallyWrittenCode}}
        // ...
        // insert your customized code here!
        // ...
    // {{ProtectedRegionEnd}}
}
```

include template `Constructor.jgt`

Template file: `Constructor.jgt`

```
/**
 * Constructor with all attributes.
 */
public <<ClassName>>(<<foreachAttribute --> Argument.jgt /,>>)
{
    << foreachAttribute --> AttributeInit.jgt />>
}
```

Template file: `Attribute.jgt`

```
protected << <-- AttributeDeclaration.jgt >>;
```

Template file: `AttributeDeclaration.jgt`

```
<<DataType>> <<AttributeName>>
```

include template `AttributeDeclaration.jgt`

Template file: `Getter.jgt`

```
/**
 * Returns the value of type <<DataType>> for attribute <<AttributeName>>.
 * @return value of <<AttributeName>>
 */
public <<DataType>> get<<+AttributeName>>()
{
    return this.<<AttributeName>>;
}
```

Includes & sub templates are useful to keep templates simple & readable!

3. Java Class Template (3)

Template file: `Constructor.jgt`

```
/**
 * Constructor with all attributes.
 */
public <<ClassName>>(<<foreachAttribute --> Argument.jgt /,>>)
{
    << foreachAttribute --> AttributeInit.jgt />>
}
```

`/,` text operator deletes the last comma which remains after the last argument

Template file: `Argument.jgt`

```
<< <-- AttributeDeclaration.jgt >>,
```

Template file: `AttributeDeclaration.jgt`

```
<<DataType>> <<AttributeName>>
```

include template `AttributeDeclaration.jgt`

Template file: `AttributeInit.jgt`

```
this.<<AttributeName>> = <<AttributeName>>;
```

Includes are quite suitable to re-use template parts multiple times!

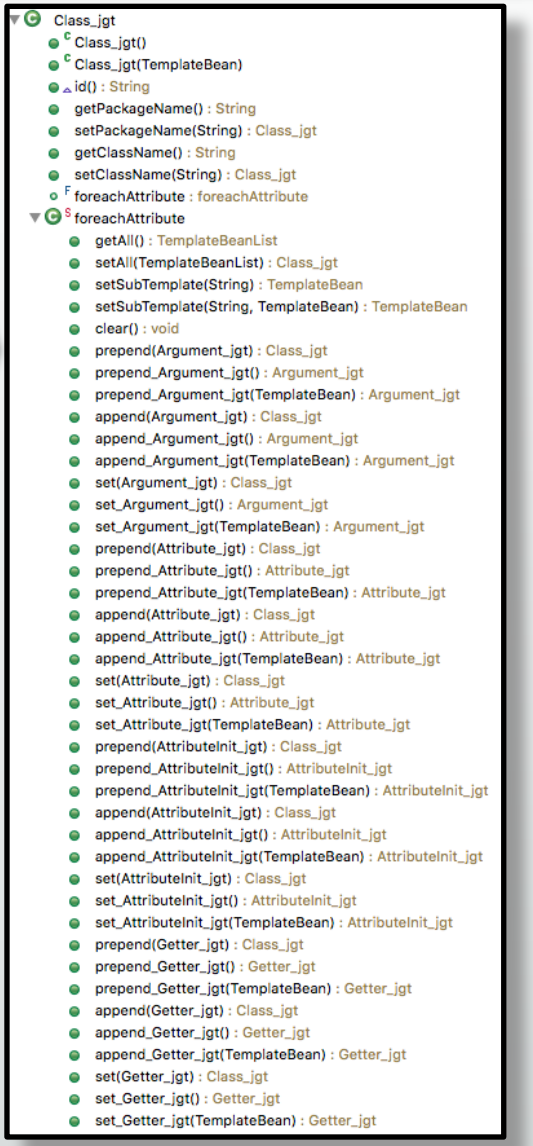
3. Java Class Template (4)

Template file: `Class.jgt`

```
package <<PackageName>>;

public class <<ClassName>>
{
    << foreachAttribute --> Attribute.jgt />>

    << Include <-- Constructor.jgt />>

    << foreachAttribute --> Getter.jgt />>

    // {{ProtectedRegionStart::ManuallyWrittenCode}}
        // ...
        // insert your customized code here!
        // ...
    // {{ProtectedRegionEnd}}
}
```

Automatic Build

Template Bean: `Class_jgt`



3. Java Class Template: Code Generator

Code Generator:

```
public void generate()
{
    // Class:
    Class_jgt t = new Class_jgt();
    t.setPackageName("example");
    t.setClassName("MyClass");

    // Attributes:
    Attribute_jgt attr1 = t.foreachAttribute.append_Attribute_jgt().setDataType("Long").setAttributeName("number");
    Attribute_jgt attr2 = t.foreachAttribute.append_Attribute_jgt().setDataType("String").setAttributeName("text");

    // Constructor arguments:
    t.foreachAttribute.append_Argument_jgt(attr1); // 'parent constructor' for variable values
    t.foreachAttribute.append_Argument_jgt(attr2); // works via automatic value propagation

    // Attribute initializations:
    t.foreachAttribute.append_AttributeInit_jgt(attr1); // 'parent constructor' for variable values
    t.foreachAttribute.append_AttributeInit_jgt(attr2); // is similar to a copy constructor

    // Getter:
    t.foreachAttribute.append_Getter_jgt(attr1);
    t.foreachAttribute.append_Getter_jgt(attr2);

    updateSourceFile("java/" + t.getPackageName().replace('.', '/') + "/" + t.getClassName() + ".java", t.toString());
}
```

We insert the attribute values only once! For all other structures (`Argument.jgt`, `AttributeInit.jgt` & `Getter.jgt`) we just use the "parent constructor". Values will be propagated automatically!

3. Java Class Template: Output

Code Generator Output:

```java
package example;

public class MyClass
{
    protected Long number;
    protected String text;

    /**
     * Constructor with all attributes.
     */
    public MyClass(Long number, String text)
    {
        this.number = number;
        this.text = text;
    }

    /**
     * Returns the value of type Long for attribute number.
     * @return value of number
     */
    public Long getNumber()
    {
        return this.number;
    }

    /**
     * Returns the value of type String for attribute text.
     * @return value of text
     */
    public String getText()
    {
        return this.text;
    }

    // {{ProtectedRegionStart::ManuallyWrittenCode}}
        // ...
        // insert your customized code here!
        // ...
    // {{ProtectedRegionEnd}}
}
```

Java Class with Inline Templates

4. Java Class with Inline Templates

Template file:

`JavaClass.jgt`

```
package <<PackageName>>;

public class <<ClassName>>
{
    << foreachAttribute --> Attribute:
    {
        protected <<DataType>> <<AttributeName>>;
    }
    />>

    /**
     * Constructor with all attributes.
     */
    public <<ClassName>>(<<foreachAttribute --> Argument: {<<DataType>> <<AttributeName>>, } />,>>)
    {
        << foreachAttribute --> AttributeInit:
        {
            this.<<AttributeName>> = <<AttributeName>>;
        }
        />>
    }

    << foreachAttribute --> Getter:
    {
        /**
         * Returns the value of type <<DataType>> for attribute <<AttributeName>>.
         * @return result value
         */
        public <<DataType>> get<<+AttributeName>>()
        {
            return this.<<AttributeName>>;
        }

    }
    />>

    // {{ProtectedRegionStart::ManuallyWrittenCode}}
        // ...
        // insert your customized code here!
        // ...
    // {{ProtectedRegionEnd}}
}
```

TemplateBean Class:

`JavaClass_jgt`

Inline templates are represented by nested template beans within the enclosing template bean!

Automatic Build

4. Java Class with Inline Templates: Code Generator

Code Generator: *Value propagation from the previous Java class template!*

```
public void generate()
{
    // Java class with includes and external sub templates
    Class_jgt t = new Class_jgt();
    t.setPackageName("example");
    t.setClassName("MyClass");

    // Attributes:
    Attribute_jgt attr1 = t.foreachAttribute.append_Attribute_jgt().setDataType("Long").setAttributeName("number");
    Attribute_jgt attr2 = t.foreachAttribute.append_Attribute_jgt().setDataType("String").setAttributeName("text");


    .
    .  // further data insertions as before
    .

    // Java class with inline sub templates:
    JavaClass_jgt c = new JavaClass_jgt(t); // 'parent constructor' to get data via value propagation
    c.setPackageName("example.inline");     // for complete template structure

    updateSourceFile("java/" + c.getPackageName().replace('.', '/') + "/" + c.getClassName() + ".java", c );
}
```

4. Java Class with Inline Templates: Output

Code Generator Output:

```java
package example.inline;

public class MyClass
{
    protected Long number;
    protected String text;

    /**
     * Constructor with all attributes.
     */
    public MyClass(Long number, String text)
    {
        this.number = number;
        this.text = text;
    }

    /**
     * Returns the value of type Long for attribute number.
     * @return value of number
     */
    public Long getNumber()
    {
        return this.number;
    }

    /**
     * Returns the value of type String for attribute text.
     * @return value of text
     */
    public String getText()
    {
        return this.text;
    }

    // {{ProtectedRegionStart::ManuallyWrittenCode}}
        // ...
        // insert your customized code here!
        // ...
    // {{ProtectedRegionEnd}}
}
```

4. Java Class with Inline Templates: Data Insertion the other Way around

Code Generator: *Data insertion the other way around!*

```
public void generate()
{
    JavaClass_jgt c = new JavaClass_jgt(t);
    c.setPackageName("example.inline");
    c.setClassName("MyClass");

    // Attributes:
    Attribute a1 = c.foreachAttribute.append_Attribute().setDataType("Long").setAttributeName("number");
    Attribute a2 = c.foreachAttribute.append_Attribute().setDataType("String").setAttributeName("text");

    // Constructor arguments:
    c.foreachAttribute.append_Argument(a1); // 'parent constructor' for variable values
    c.foreachAttribute.append_Argument(a2); // works via automatic value propagation

    // Attribute initializations:
    c.foreachAttribute.append_AttributeInit(a1); // 'parent constructor' for variable values
    c.foreachAttribute.append_AttributeInit(a2); // works via automatic value propagation

    // Getter:
    c.foreachAttribute.append_Getter(a1); // 'parent constructor' for variable values
    c.foreachAttribute.append_Getter(a2); // works via automatic value propagation

    updateSourceFile("java/" + c.getPackageName().replace('.', '/') + "/" + c.getClassName() + ".java", c );

    // Java class with external sub templates:
    Class_jgt t = new Class_jgt(c);  // value propagation the other way around!
    t.setPackageName("example");     // moving this class into another package

    updateSourceFile("java/" + t.getPackageName().replace('.', '/') + "/" + t.getClassName() + ".java", t );
}
```

The code generator output is identical to the previous slide(s)!

5. Automatic Value Propagation: Summary

Summary: value propagation for sub templates

- Value propagation for sub templates only takes place if the list in the child template bean has not been set, i.e. if it is null.
- For each sub template bean ***a*** of the parent bean, an instance of the associated sub template bean ***b*** of the child bean is created with ***a*** as its parent.
- Mapping between sub template beans of different types (but similar names) is performed via their name prefix.
- An external sub template `Attribute.jgt` (prefix "Attribute") of the parent bean is mapped to another external sub template, like `Attribute.cgt`, for example (where `.cgt` might have been chosen as suffix for C++ templates).
- If there is no appropriate external sub template it will be mapped to an inline sub template (`Attribute`, for example) with same prefix (if there is one).
- The reverse mapping approach is taken, if the sub template bean to be mapped is an inline sub template bean (➠ first inline, then external).

APPENDIX

APPENDIX

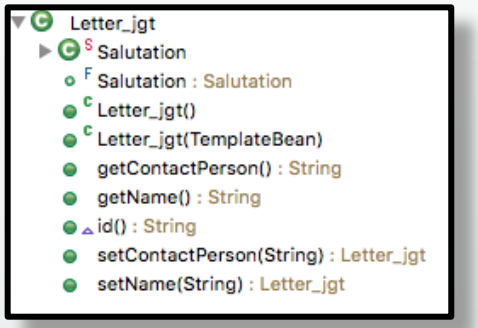
Multilingual Templates: Three Languages at Once

Template file: `Letter.jgt`

```
Dear <<Salutation --> Mr: {Mr.} | Mrs: {Mrs.}>> <<Name>>,

we hereby want to inform you...
bla bla bla .

Best regards,

<<ContactPerson>>
```

Automatic Build

TemplateBean Class: `Letter_jgt`

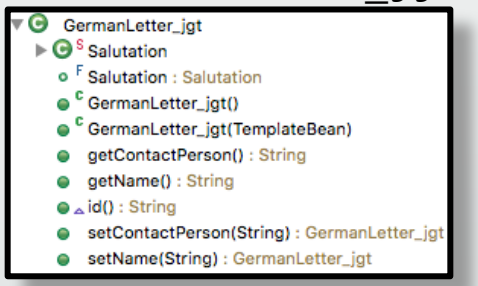


Template file: `GermanLetter.jgt`

```
Sehr <<Salutation --> Mr: {geehrter Herr} | Mrs: {geehrte Frau}>> <<Name>>,

wir möchten Sie hiermit darauf hinweisen...
bla bla bla ...

Mit freundlichen Grüßen

<<ContactPerson>>
```

Automatic Build

TemplateBean Class: `GermanLetter_jgt`

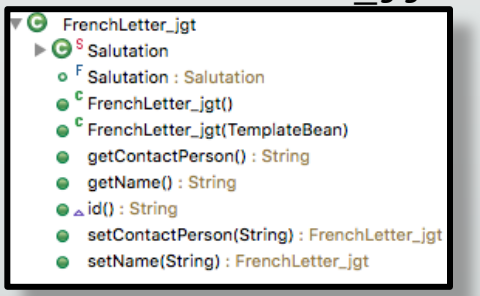


Template file: `FrenchLetter.jgt`

```
<<Salutation --> Mr: {Cher Monsieur} | Mrs: {Chère Madame}>> <<Name>>,

nous aimerions vous signaler...
bla bla bla...

Avec nous meilleurs sentiments,

<<ContactPerson>>
```

Automatic Build

TemplateBean Class: `FrenchLetter_jgt`



Multilingual Templates: Three Languages at Once (2)

```
Letter_jgt letter = new Letter_jgt();
letter.Salutation.set_Mr();
letter.setName("Bean").setContactPerson("Mary Moneypenny");

// two examples for automatic value propagation
// via 'parent constructor' initialization:
GermanLetter_jgt brief  = new GermanLetter_jgt(letter);
FrenchLetter_jgt lettre = new FrenchLetter_jgt(letter);

System.out.println(letter);
System.out.println(brief);
System.out.println(lettre);
```

```
Dear Mr. Bean,

we hereby want to inform you...
bla bla bla .

Best regards,

Mary Moneypenny
```

```
Sehr geehrter Herr Bean,

wir möchten Sie hiermit darauf hinweisen...
bla bla bla .

Mit freundlichen Grüßen

Mary Moneypenny
```

```
Cher Monsieur Bean,

nous aimerions vous signaler...
bla bla bla .

Avec nous meilleurs sentiments,

Mary Moneypenny
```

This is called automatic value propagation!

It works for variable values as well as for sub template instances!

Questions ?

Feedback ?

codegen@jiowa.de

Java Doc: www.jiowa.de/jiowa-codegen/doc/api/

PDF: www.jiowa.de/jiowa-codegen/doc/Jiowa-Value-Propagation-in-Code-Generator-Templates.pdf