JIOWA

EFFICIENCY SIMPLICITY FLEXIBILITY RELIABILITY

JIOWA Code Generation Framework

Tutorial & Handbook for the Code Generation Framework and its Template Engine

Dr. Robert Mencl
Independent Consultant / www.mencl.de

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

codegen@jiowa.de
www.jiowa.de

Version 2.1.2, March 16th, 2016
www.jiowa.de/download.html

JIOWA Code Generation Framework

1. Simple Examples

Email Template

1. Simple Examples: 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

<< ... >>

1. Simple Examples: 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>>
```

Plain text enriched with

template notation elements

which are enclosed by

<< ... >>

- red symbols: Template notation elements.
- green text: Identifiers which can be chosen arbitrarily by the user!!!

  These identifiers will be used by the system when generating

  so-called *TemplateBeans* out of these template files!

1. Simple Examples: Email Template (3)

subtemplate structure identifier: Salutation

inline template identifiers: Mr and Mrs and : is the qualifier symbol

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

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

Best regards,

<<ContactPerson>>
```

variable: <<Name>>

inline templates: {Mr.} and {Mrs.}

variable: <<ContactPerson>>

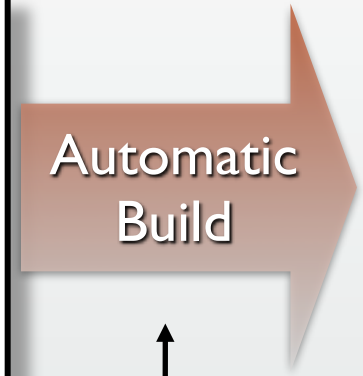
--> is the subtemplate indicator and | is the or-symbol

for different possible sub templates.

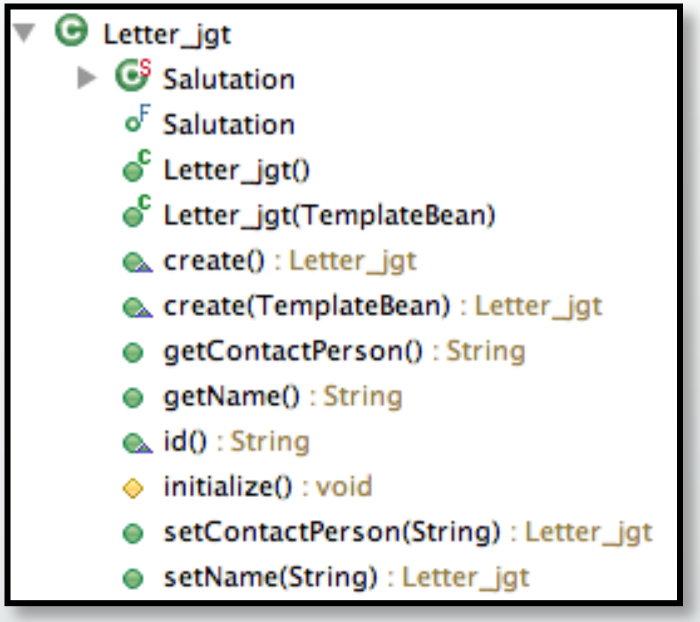
1. Simple Examples: Email Template (4)

Template file: `Letter.jgt`

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

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

Best regards,

<<ContactPerson>>
```



TemplateBean Class: `Letter_jgt`



```
Letter_jgt
  Salutation
  Salutation
  Letter_jgt()
  Letter_jgt(TemplateBean)
  create() : Letter_jgt
  create(TemplateBean) : Letter_jgt
  getContactPerson() : String
  getName() : String
  id() : String
  initialize() : void
  setContactPerson(String) : Letter_jgt
  setName(String) : Letter_jgt
```

compile-time safety

for template syntax

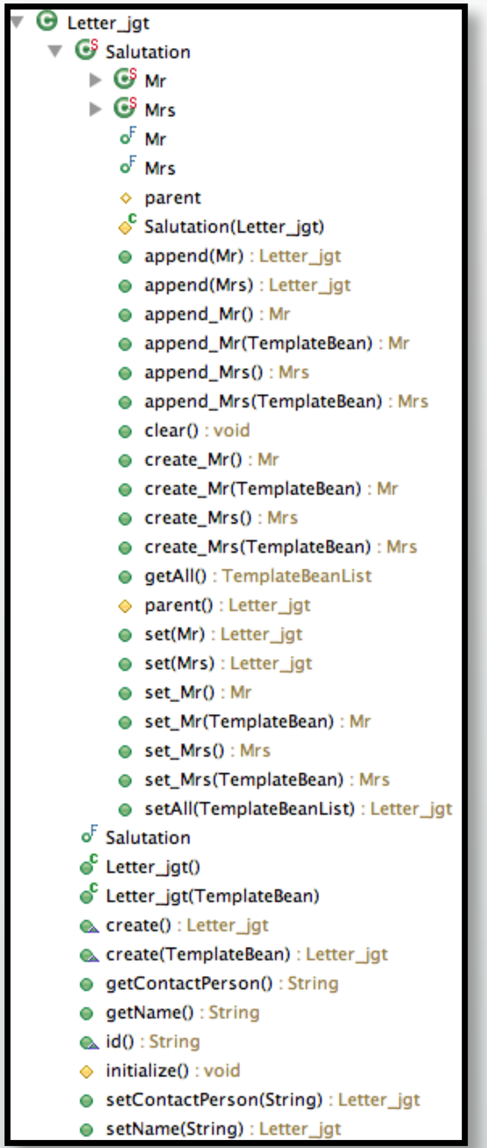
1. Simple Examples: Email Template (5)

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`



```
▼ Letter_jgt
    ▼ Salutation
        ▶ Mr
        ▶ Mrs
          Mr
          Mrs
          parent
          Salutation(Letter_jgt)
          append(Mr) : Letter_jgt
          append(Mrs) : Letter_jgt
          append_Mr() : Mr
          append_Mr(TemplateBean) : Mr
          append_Mrs() : Mrs
          append_Mrs(TemplateBean) : Mrs
          clear() : void
          create_Mr() : Mr
          create_Mr(TemplateBean) : Mr
          create_Mrs() : Mrs
          create_Mrs(TemplateBean) : Mrs
          getAll() : TemplateBeanList
          parent() : Letter_jgt
          set(Mr) : Letter_jgt
          set(Mrs) : Letter_jgt
          set_Mr() : Mr
          set_Mr(TemplateBean) : Mr
          set_Mrs() : Mrs
          set_Mrs(TemplateBean) : Mrs
          setAll(TemplateBeanList) : Letter_jgt
      Salutation
      Letter_jgt()
      Letter_jgt(TemplateBean)
      create() : Letter_jgt
      create(TemplateBean) : Letter_jgt
      getContactPerson() : String
      getName() : String
      id() : String
      initialize() : void
      setContactPerson(String) : Letter_jgt
      setName(String) : Letter_jgt
```

Using the template bean in your program:

```java
Letter_jgt template = new Letter_jgt();

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

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

1. Simple Examples: Email Template (6)

Template file: `Letter.jgt`

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

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

Best regards,

<<ContactPerson>>
```

Using the template bean in your program:

```java
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
```

1. Simple Examples

Java Class Template

1. Simple Examples: Java Class Template

Template file: `Class.jgt` *(just a simple JavaBean with getters & setters)*

```
package <<PackageName>>;

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

    <<foreachAttribute --> GetterSetter.jgt >>
}
```

Have you ever seen simpler templates for java classes?

Template file: `Attribute.jgt`

```
protected <<DataType>> <<AttributeName>>;
```

syntax highlighting for the target platform and **not** for the template notation !

Template file: `GetterSetter.jgt`

```
public <<DataType>> get<<+AttributeName>>()
{
    return this.<<AttributeName>>;
}

public void set<<+AttributeName>>(<<DataType>> value)
{
    this.<<AttributeName>> = value;
}
```

template notation does not change the visual appearance of the code too much !

`<<+AttributeName>>` means toFirstUpperCase() of AttributeName

1. Simple Examples: Java Class Template (2)

Template file: `Class.jgt`

```
package <<PackageName>>;

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

    <<foreachAttribute --> GetterSetter.jgt >>
}
```

`foreachAttribute` *does not perform any kind of iteration!!! It is an arbitrarily chosen word!*

Code Example:

```
Class_jgt template = new Class_jgt();

// example iteration over some attributes
for (SomeAttributeClass attr : someAttributeList )
{
   Attribute_jgt attribute = new Attribute_jgt().
                                setDataType(attr.getDataTypeName()).
                                setAttributeName(attr.getName());

   GetterSetter_jgt gettersetter = new GetterSetter_jgt(attribute);

   template.foreachAttribute.append(attribute).append(gettersetter);
}
```

This code would be part of your code generator.

The iteration over the attributes of a class has to be performed in the code generator which is written in pure Java.

Variable values of GetterSetter *are getting initialized via copy constructor.*

1. Simple Examples: Java Class Template (3)

Template file: `Class.jgt`

```
package <<PackageName>>;

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

    <<foreachAttribute --> GetterSetter.jgt >>
}
```

Template file: `Attribute.jgt`

```
protected <<DataType>> <<AttributeName>>;
```

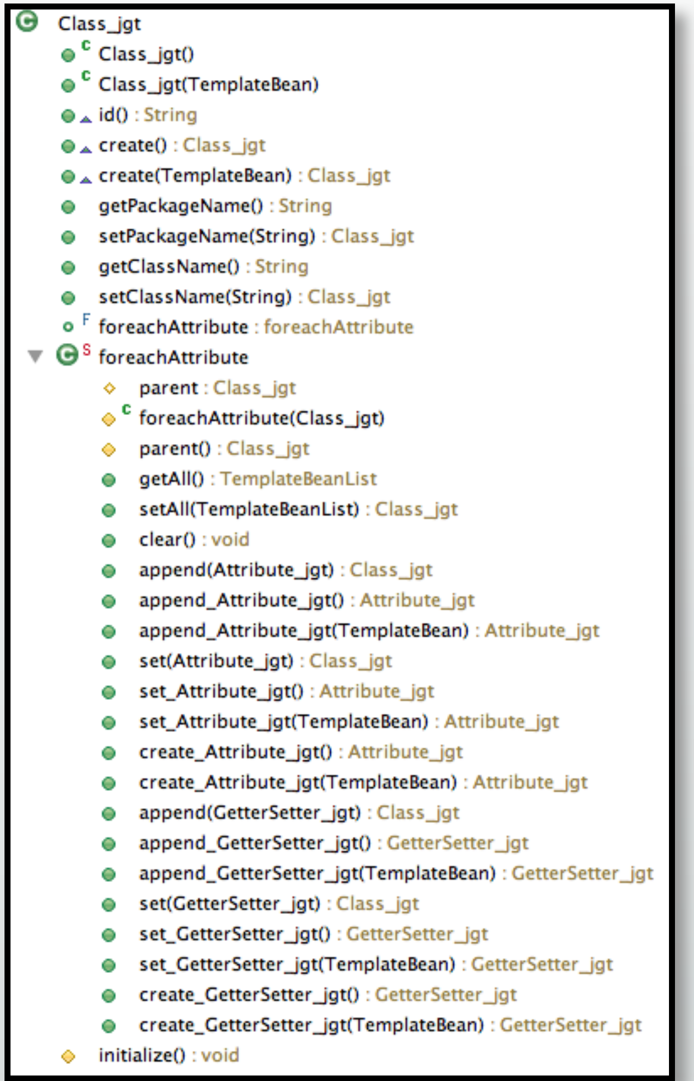
Template file: `GetterSetter.jgt`

```
public <<DataType>> get<<+AttributeName>>()
{
    return this.<<AttributeName>>;
}

public void set<<+AttributeName>>(<<DataType>> value)
{
    this.<<AttributeName>> = value;
}
```

Automatic Build

compile-time safety for template syntax

TemplateBean Class: `Class_jgt`



```
Class_jgt
    Class_jgt()
    Class_jgt(TemplateBean)
    id() : String
    create() : Class_jgt
    create(TemplateBean) : Class_jgt
    getPackageName() : String
    setPackageName(String) : Class_jgt
    getClassName() : String
    setClassName(String) : Class_jgt
    foreachAttribute : foreachAttribute
    foreachAttribute
        parent : Class_jgt
        foreachAttribute(Class_jgt)
        parent() : Class_jgt
        getAll() : TemplateBeanList
        setAll(TemplateBeanList) : Class_jgt
        clear() : void
        append(Attribute_jgt) : Class_jgt
        append_Attribute_jgt() : Attribute_jgt
        append_Attribute_jgt(TemplateBean) : Attribute_jgt
        set(Attribute_jgt) : Class_jgt
        set_Attribute_jgt() : Attribute_jgt
        set_Attribute_jgt(TemplateBean) : Attribute_jgt
        create_Attribute_jgt() : Attribute_jgt
        create_Attribute_jgt(TemplateBean) : Attribute_jgt
        append(GetterSetter_jgt) : Class_jgt
        append_GetterSetter_jgt() : GetterSetter_jgt
        append_GetterSetter_jgt(TemplateBean) : GetterSetter_jgt
        set(GetterSetter_jgt) : Class_jgt
        set_GetterSetter_jgt() : GetterSetter_jgt
        set_GetterSetter_jgt(TemplateBean) : GetterSetter_jgt
        create_GetterSetter_jgt() : GetterSetter_jgt
        create_GetterSetter_jgt(TemplateBean) : GetterSetter_jgt
    initialize() : void
```

1. Simple Examples: Java Class Template (4)

Class.jgt

```
package <<PackageName>>;

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

    <<foreachAttribute --> GetterSetter.jgt >>
}
```

Attribute.jgt

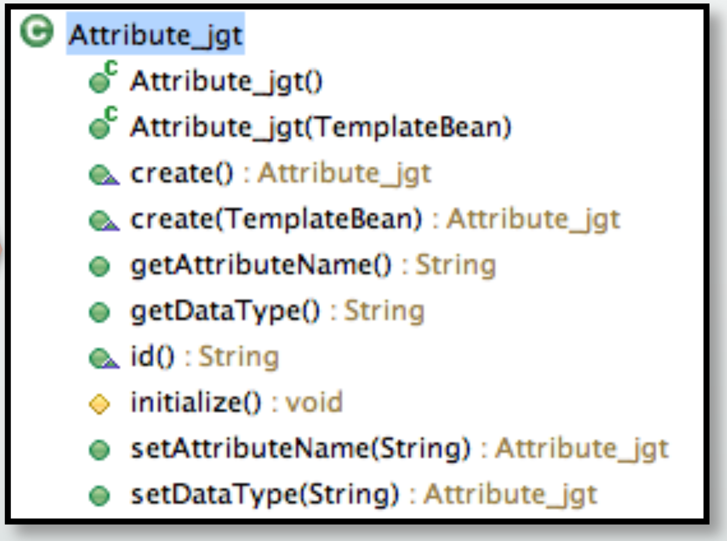
```
protected <<DataType>> <<AttributeName>>;
```

GetterSetter.jgt

```
public <<DataType>> get<<+AttributeName>>()
{
    return this.<<AttributeName>>;
}

public void set<<+AttributeName>>(<<DataType>> value)
{
    this.<<AttributeName>> = value;
}
```

Automatic Build

TemplateBean Class: Attribute_jgt



```
Attribute_jgt
    Attribute_jgt()
    Attribute_jgt(TemplateBean)
    create() : Attribute_jgt
    create(TemplateBean) : Attribute_jgt
    getAttributeName() : String
    getDataType() : String
    id() : String
    initialize() : void
    setAttributeName(String) : Attribute_jgt
    setDataType(String) : Attribute_jgt
```

1. Simple Examples: Java Class Example (5)

Class.jgt

```
package <<PackageName>>;

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

    <<foreachAttribute --> GetterSetter.jgt >>
}
```

Attribute.jgt

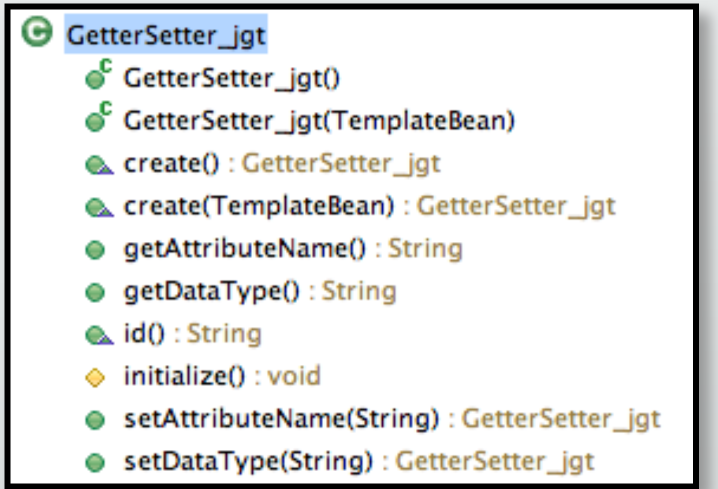
```
protected <<DataType>> <<AttributeName>>;
```

GetterSetter.jgt

```
public <<DataType>> get<<+AttributeName>>()
{
    return this.<<AttributeName>>;
}

public void set<<+AttributeName>>(<<DataType>> value)
{
    this.<<AttributeName>> = value;
}
```

Automatic Build

TemplateBean Class: GetterSetter_jgt



```
GetterSetter_jgt
    GetterSetter_jgt()
    GetterSetter_jgt(TemplateBean)
    create() : GetterSetter_jgt
    create(TemplateBean) : GetterSetter_jgt
    getAttributeName() : String
    getDataType() : String
    id() : String
    initialize() : void
    setAttributeName(String) : GetterSetter_jgt
    setDataType(String) : GetterSetter_jgt
```

1. Simples Examples: Java Class Example (6)

Models to be used: ***... many possibilities ...***

1. Graphical Model:
   - Examples: UML, BPMN, ...
2. Textual Model:
   - Java classes (can be analyzed to generate additional technical layer with the code generator),
   - domain specific languages (DSL),
   - anything can be a used for the description of a model (ASCII Text, MS-Word, PDF, ...).

You just need a Java API to read the data from your model instance!

1. Simples Examples: Java Class Example (7)

Graphical Model:

- UML:
  - Arbitrary UML Modeling Tools: use **Jiowa UML-API** for Java.
    - It is part of **Jiowa-MDSD** software package *(to be published soon)*
    - `jiowa-mdsd = { jiowa-codegen , jiowa-uml }`
    - Support for Enterprise Architect, Visual Paradigm, MagicDraw, IBM RSA, Altova UModel, Eclipse UML, etc. *Compensates the deviations between the different export format dialects (XMI) of the tools.*
    - ***can automatically learn new UML formats*** and adapt the API to your specific modeling tool (automated *specification by example*).
  - Eclipse-based UML/EMF: use Java-UML-API from the eclipse project.
- Any other graphical notation:
  - you will just need a Java API to read data from your model instance.

1. Simple Examples: Java Class Example (8)

Example for UML model access with the Jiowa UML-API (not part of this distribution)

to be published soon in the ***jiowa-mdsd*** *distribution*

```
public class MyGenerator extends AbstractGenerator
{
   .
   .
   .
    @Override
    public void generate()
    {
        for (UMLClass umlClass : umlNavigator.getUMLClassListByStereotypeName("Example"))
        {
            Class_jgt template = new Class_jgt().setPackageName("example").setClassName(umlClass.getName());

            for (UMLProperty umlAttr : umlClass.getOwnedAttributeList())
            {
                Attribute_jgt attribute = new Attribute_jgt();
                attribute.setDataType(umlAttr.getDataTypeName()).setAttributeName(umlAttr.getName());

                GetterSetter_jgt gettersetter = new GetterSetter_jgt(attribute);

                template.foreachAttribute.append(attribute).append(gettersetter);
            }

            String filepath = template.getPackageName().replace('.', '/')
                                  + "/" + template.getClassName() + ".java";
            updateSourceFile ( filepath , template.toString() ) ;
        }
    }
}
```

generate code

for all classes with

stereotype „Example“

copy constructor
for attributes

1. Simples Examples: Java Class Example (9)

Model Access:

- If you have a **textual model**:
  - **Java classes:** analyze them and generate additional classes.
  - **Text files:** Textual Models can also be simple text files, MS-Word & PDF documents, etc... It just depends on your interpretation (meta knowledge) of the contents of these documents.
  - **Textual DSL:** use the generated Java API to read data from your domain specific model instance.

➔ *you only need a Java API to read the data from your model instances.*

➔ *since the templates do not depend on any specific model type or structure you can choose any type of model.*

1. Simple Examples: Java Class by using Inline Templates

Java Class with Inline Templates

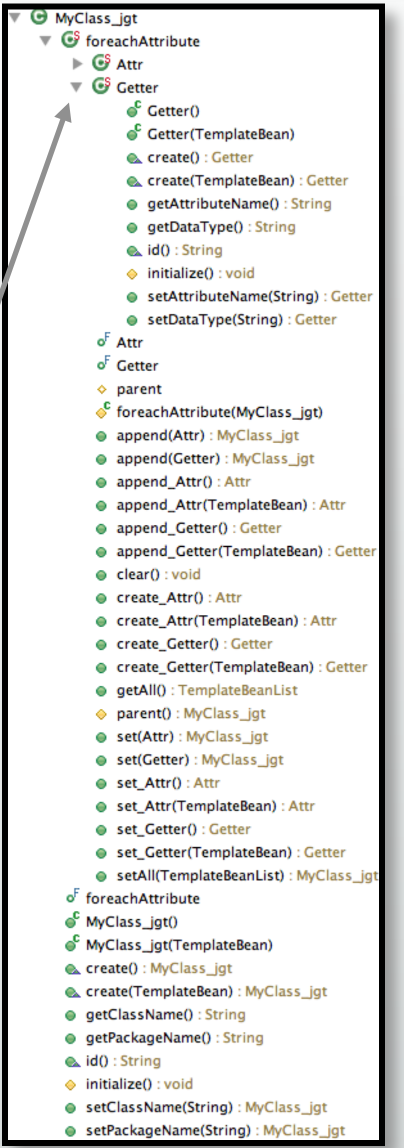
1. Simple Examples: Java Class by using Inline Templates

Template file: `MyClass.jgt`

```
package <<PackageName>>;

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

    <<foreachAttribute --> Getter:
    {
        public <<DataType>> get<<+AttributeName>>()
        {
            return this.<<AttributeName>>;
        }

        public void set<<+AttributeName>>(<<DataType>> value)
        {
            this.<<AttributeName>> = value;
        }
    }
    >>
}
```

Automatic Build

TemplateBean Class: `MyClass_jgt`



```
▼ MyClass_jgt
    ▼ foreachAttribute
        ▶ Attr
        ▼ Getter
            Getter()
            Getter(TemplateBean)
            create() : Getter
            create(TemplateBean) : Getter
            getAttributeName() : String
            getDataType() : String
            id() : String
            initialize() : void
            setAttributeName(String) : Getter
            setDataType(String) : Getter
        Attr
        Getter
        parent
        foreachAttribute(MyClass_jgt)
        append(Attr) : MyClass_jgt
        append(Getter) : MyClass_jgt
        append_Attr() : Attr
        append_Attr(TemplateBean) : Attr
        append_Getter() : Getter
        append_Getter(TemplateBean) : Getter
        clear() : void
        create_Attr() : Attr
        create_Attr(TemplateBean) : Attr
        create_Getter() : Getter
        create_Getter(TemplateBean) : Getter
        getAll() : TemplateBeanList
        parent() : MyClass_jgt
        set(Attr) : MyClass_jgt
        set(Getter) : MyClass_jgt
        set_Attr() : Attr
        set_Attr(TemplateBean) : Attr
        set_Getter() : Getter
        set_Getter(TemplateBean) : Getter
        setAll(TemplateBeanList) : MyClass_jgt
    foreachAttribute
    MyClass_jgt()
    MyClass_jgt(TemplateBean)
    create() : MyClass_jgt
    create(TemplateBean) : MyClass_jgt
    getClassName() : String
    getPackageName() : String
    id() : String
    initialize() : void
    setClassName(String) : MyClass_jgt
    setPackageName(String) : MyClass_jgt
```

Inline Templates are represented by nested classes within the TemplateBean

1. Simple Examples: Java Class by using Inline Templates (2)

Example for model access with the Jiowa UML-API (not part of this distribution)

*to be published soon in the **jiowa-mdsd** distribution*

```
public class MyGenerator extends AbstractGenerator
{
  .
  .
  .
    @Override
    public void generate()
    {
        for (UMLClass umlClass : umlNavigator.getUMLClassListByStereotypeName("Example"))
        {
            MyClass_jgt template = new MyClass_jgt().setPackageName("example").setClassName(umlClass.getName());

            for (UMLProperty umlAttr : umlClass.getOwnedAttributeList())
            {
                MyClass_jgt.foreachAttribute.Attr attr =
                    template.foreachAttribute.append_Attr().
                        setDataType(umlAttr.getDataTypeName()).
                             setAttributeName(umlAttr.getName());

                template.foreachAttribute.append_Getter(attr);
            }

            String filepath = template.getPackageName().replace('.', '/')
                                  + "/" + template.getClassName() + ".java";
            updateSourceFile ( filepath , template.toString() ) ;
        }
    }
}
```

Access to inline templates!

copy constructor for attributes

Features

Features of the JIOWA Code Generator Framework:

- compile-time safety for your templates,
- full control of code generation via Java,
- very effective & comprehensible template notation,
- strict separation of code (Java generator) & design (template),
- templates **do not** carry any model specific information
  ➠ completely re-usable across different projects,
- supports any type of model for which an API exists,
- supports each IDE (no plug-ins necessary),
- easily extensible via Java,
- **no** polyglot programming.

2. Template Notation

Template Notation

2. Template Notation

Characteristics of the JIOWA Template Notation:

- only *key symbols*: `<< --> : { | } >> ... <-- + - / _ = {[( , ==> )]} \ //#` **No keywords!**
- basically just *two structures*:
  - variables: `<<VariableName>>`
  - and subtemplate structures:

    `<<foreachElement --> SubTemplate.jgt>>`

  ... plus some variations: *inline templates* and *variable & text operators*

}

... this is enough to create any kind of text file quite easily.

- templates are compiled into Template Beans (POJOs) via automatic build process of your IDE:
  - switch on automatic Maven build,
  - (optionally) edit properties in `jiowa.codegen.properties`
    (default configuration works out-of-the-box !)

2. Template Notation: Comments

Comments:

```
//# deletes the whole line within the created template text

    //# keeps the first 4 characters and deletes the rest of the line
```

`//#` is the only key symbol for comments in the template notation

2. Template Notation: Variable Operators

Variable Operators:

```
<<+ClassName>>        //# same as toFirstUpperCase()
<<++ClassName>>       //# same as toUpperCase()
<<+++ClassName>>      //# same as toCamelCase(): transform all String Tokens into camel case
                      //# (i.e. to alternating upper and lower case;
                      //# string token delimiters are spaces and special characters )
<<-ClassName>>        //# same as toFirstLowerCase()
<<--ClassName>>       //# same as toLowerCase()
<<_ClassName>>        //# replace all spaces with _
<<__ClassName>>       //# replace all non-alphanumeric chars with _
<<__/ClassName>>      //# remove all non-alphanumeric chars
<<_/ClassName>>       //# remove all spaces
<<+_/ClassName>>      //# remove all spaces and then toFirstUpperCase()

<<+--_/ClassName>>        //# remove all spaces, then toLowerCase() and then toFirstUpperCase()
<< + -- _/ ClassName>>    //# same as above
```

- *Operators are applied (mathematically) from right to left onto the variable value.*
- *You can combine as many operators as you want,*
- *and separate them with spaces if necessary (because parsing works from left to right.)*

2. Template Notation: Variable Operators (2)

Character Replacement Operator for Variables:

```
//# replacement operator for variable values:

<<.=:ClassName>>     // replace each . within the class name with :
<<,=;ClassName>>     // replace each colon , with ;
<<.= ClassName>>     // replace each . with one whitespace
<<.=ClassName>>      // replace each . with C
                     // the variable name is just ‘lassName‘
<< ==ClassName>>     // replace each whitespace with a =
<<== ClassName>>     // replace each = with a whitespace


<<a=bd=fClassName>>    // replace a with b and d with f
<<a=b d=f ClassName>>  // same as above
```

2. Template Notation: Sub Templates

Sub Templates:

```
<<foreachElement --> SubTemplate.jgt>>
```

This structure allows the insertion of an arbitrary list of instances of the `SubTemplate_jgt` *template bean class.*

Possible examples for filling data into the bean:

```
template.foreachElement.append(new SubTemplate_jgt()); // possible
template.foreachElement.append_SubTemplate_jgt();  // possible as well

// inserts exactly one element into the list and removes everything else
template.foreachElement.set_Second_jgt();

// setting variables:
template.foreachElement.append_Second_jgt().setClassName(“MyClass");
// setting variables in a generic way (independent from the template bean type):
template.foreachElement.append_Second_jgt().set("ClassName“, "MyClass");
```

... see in JavaDoc of generated TemplateBeans for many other methods

to create, modify & append data !

2. Template Notation: Sub Templates (2)

Language independence of substructure qualifiers:

Since the template notation is based only on key symbols and not on keywords, you can use any text of any language for the qualifiers...

substructure qualifier

German:

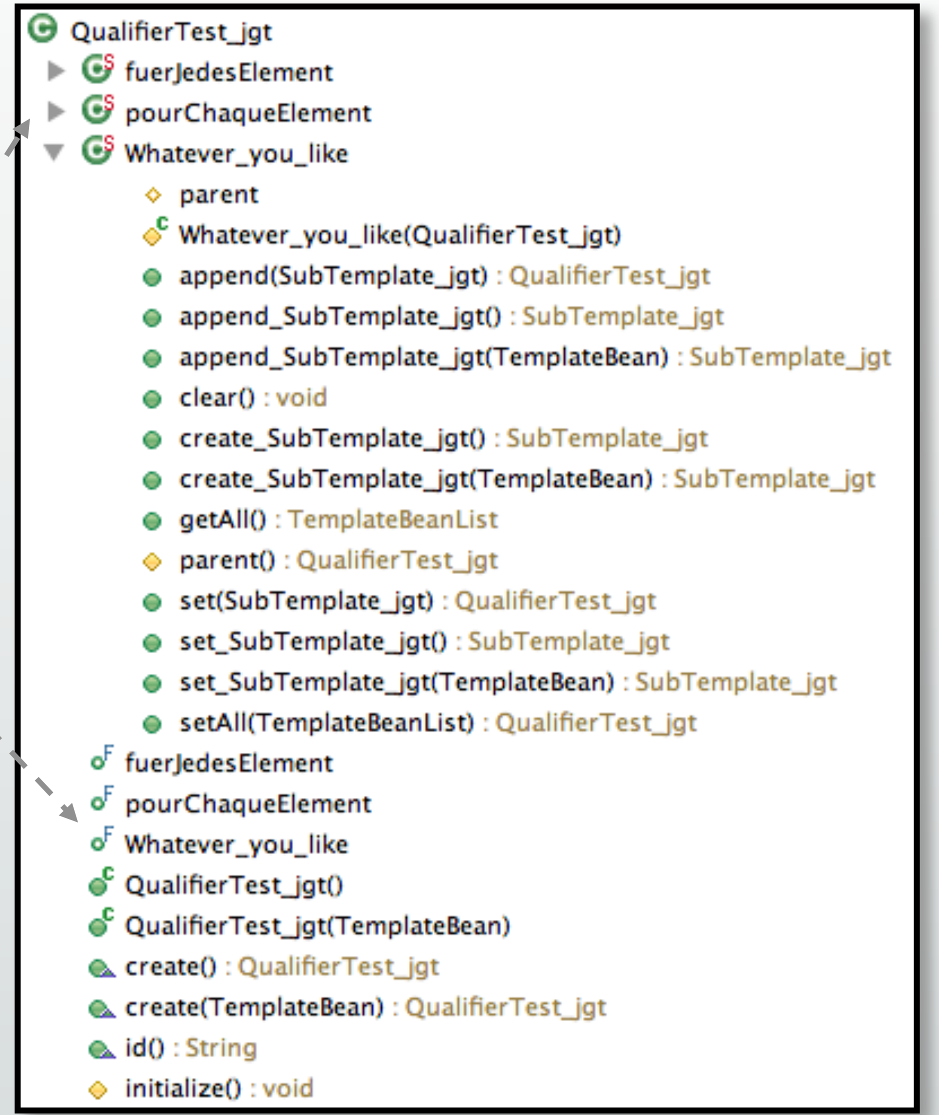
```
<< fuerJedesElement --> SubTemplate.jgt>>
```

French:

```
<< pourChaqueElement --> SubTemplate.jgt>>
```

Free Text:

```
<< Whatever you like --> SubTemplate.jgt>>
```



```
QualifierTest_jgt
  ▶ fuerJedesElement
  ▶ pourChaqueElement
  ▼ Whatever_you_like
        parent
        Whatever_you_like(QualifierTest_jgt)
        append(SubTemplate_jgt) : QualifierTest_jgt
        append_SubTemplate_jgt() : SubTemplate_jgt
        append_SubTemplate_jgt(TemplateBean) : SubTemplate_jgt
        clear() : void
        create_SubTemplate_jgt() : SubTemplate_jgt
        create_SubTemplate_jgt(TemplateBean) : SubTemplate_jgt
        getAll() : TemplateBeanList
        parent() : QualifierTest_jgt
        set(SubTemplate_jgt) : QualifierTest_jgt
        set_SubTemplate_jgt() : SubTemplate_jgt
        set_SubTemplate_jgt(TemplateBean) : SubTemplate_jgt
        setAll(TemplateBeanList) : QualifierTest_jgt
    fuerJedesElement
    pourChaqueElement
    Whatever_you_like
    QualifierTest_jgt()
    QualifierTest_jgt(TemplateBean)
    create() : QualifierTest_jgt
    create(TemplateBean) : QualifierTest_jgt
    id() : String
    initialize() : void
```

2. Template Notation: Sub Templates (3)

Choosing between different sub templates:

or-symbols

In your template file:

```
<<foreachElement --> First.jgt | Second.jgt | Third.jgt>>
```

In your generator code:

```
template.foreachElement.append_Third_jgt();
template.foreachElement.append(new First_jgt());   // arbitrary order of elements
template.foreachElement.append(new Second_jgt());
template.foreachElement.append(new Third_jgt());
template.foreachElement.append_Second_jgt().setClassName("MyClass"); // sets classname
// setting variables in a generic way (independent from the template bean type):
template.foreachElement.append_Second_jgt().set("ClassName“, "MyClass");
```

Type Safety!

Only instances of `First_jgt`, `Second_jgt` and `Third_jgt` are processed!

2. Template Notation: Inline Templates

qualifier symbol :

Inline Templates:

```
public class <<ClassName>> <<ifHasSuperClass --> Extends : {extends <<SuperClassName>>}>>
{
  ... // some stuff
}
```

point of text insertion

inline template text between { ... }

inline template qualifier Extends

Filling Data into the Template Bean:

```
// set ClassName:
template.setClassName("MyClass");

// set SuperClassName:
template.ifHasSuperClass.set_Extends().setSuperClassName("MySuperClass"); // 1st possibility

// or

template.ifHasSuperClass.set(new Extends().setSuperClassName("MySuperClass")); // 2nd possibility
```

2. Template Notation: Inline Templates (2)

Switching between sub templates:

```
public class <<ClassName>> <<ifExtends --> Extends: {extends <<SuperClassName>> } <<
                              ifInterface --> Implements : {implements }         |
                                              Name:        {<<InterfaceName>> }  |
                                              Comma:       {, }                    >>
{
  ... // some stuff
}
```

Filling values into this template bean:

```
 template.setClassName("MyClass");
 template.ifExtends.set_Extends().setSuperClassName("MySuperClass");
 template.ifInterface.append_Implements();
 template.ifInterface.append_Name().setInterfaceName("FirstInterface");
 template.ifInterface.append_Comma();
 template.ifInterface.append_Name().setInterfaceName("SecondInterface");
```

2. Template Notation: Includes

Including other templates:

```
<< Include <-- A.jgt >>        //# inserts template A.jgt directly at this position
                               //# the word „Include“ is arbitrary !!!
                               //# you can write whatever you want

<< Insert here <-- A.jgt>>     //# the text left from <-- is just for informational purposes
<< <-- A.jgt>>                 //# it is merely a comment
<<<--B.jgt>>                   //# no spaces needed

<< bla bla <-- A.jgt>>
```

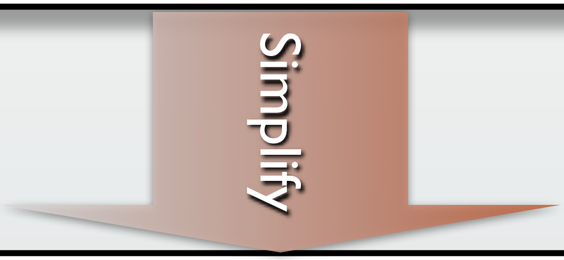
- *the text here can be chosen arbitrarily.*
- *it can be treated as a comment for other developers.*

Includes directly copy and insert the content of the included template!

2. Template Notation: Simplifying Templates

Simplifying Templates with Includes:

```
public class <<ClassName>> <<ifExtends --> Extends: {extends <<SuperClassName>> }>> <<
                              ifInterface --> Implements : {implements }          |
                                              Name:        {<<InterfaceName>> }   |
                                              Comma:       {, }                     >>
{
  ... // some stuff
}
```



```
public class <<ClassName>> << <--Extends.jgt>> << <--Implements.jgt>>
{
  ... // some stuff
}
```

Template file: `Extends.jgt`

```
<<ifExtends --> Extends: {extends <<SuperClassName>>} >>
```

Template file: `Implements.jgt`

```
<<ifInterface --> Implements: {implements } | Name: {<<InterfaceName>> } | Comma: {, } >>
```

2. Template Notation: Indenting in Inline Templates

Indenting in Inline Templates:

```
public class <<ClassName>> <<ifExtends
                  --> Extends: {extends <<SuperClassName>> }>>
{
    <<foreachAttribute --> Attr:
    {
        protected <<DataType>> <<AttributeName>>;
    }
    >>

    <<foreachAttribute --> Getter:
    {
        public <<DataType>> get<<+AttributeName>>()
        {
            return this.<<AttributeName>>;
        }

        public void set<<+AttributeName>>(<<DataType>> value)
        {
            this.<<AttributeName>> = value;
        }
    }
    >>
}
```

- **brackets { ... } in the same row:**
  the enclosed text will be copied and inserted exactly as it was written.

- **{ ...} in more than one row:**
  <u>leftmost non-whitespace character position</u> defines offset column from which all subsequent lines are adjusted.

plus one return character

Important if you want to generate visually appealing code !

2. Template Notation: Indenting in Inline Templates (2)

Indenting in Inline Templates:

point of text insertion

```
public class <<ClassName>> <<ifExtends
                           --> Extends: {extends <<SuperClassName>> }>>
{
    <<foreachAttribute --> Attr:
    {
        protected <<DataType>> <<AttributeName>>;
    }
    >>

    <<foreachAttribute --> Getter:
    {
        public <<DataType>> get<<+AttributeName>>()
        {
            return this.<<AttributeName>>;
        }

        public void set<<+AttributeName>>(<<DataType>> value)
        {
            this.<<AttributeName>> = value;
        }
    }
    >>
}
```

point of text insertion

2. Template Notation: Indenting in Inline Templates (3)

Some tricks with the indentation mechanism:

```
<<foreachElement -->
  Inline: { /* something here */ } |
  NewLine: {
           }     //# Here the text is only a line feed
  |
  EmptyText: {}
  |
  TwoNewLines: {

               }
>>
```

{...} : If it is more than one row and there are no non-whitespaces, it delivers as many LF as there are rows.

just empty text

{ is the leftmost non-whitespace character in this case.

Filling data into the template bean:

```
template.foreachElement.append_Inline();         // the order and the number is arbitrary
template.foreachElement.append_NewLine();        // adds a new line after the last insertion
template.foreachElement.append_EmptyText();      // „adds“ an empty string
template.foreachElement.append_TwoNewLines();    // „adds“ an two new line feeds
```

2. Template Notation: Multiply Nested Inline Templates

Multiply Nested Inline Templates:

```
public class <<ClassName>>
{
  ... // some program code here

   <<Implementation -->
   VersionA:
   {
     // here comes another inline template
     <<AnotherSubStructure --> VersionAA:
       {
          // some other program stuff here...
       }
   }
   |
   VersionB:
   {
      <<Again_a_SubStructure --> VersionBA:
        {
           // some program stuff here...
        }
   }
}
```

You can use multiply nested inline templates on as many hierarchical levels as you want!

2. Template Notation: Dynamic Notation Change

Dynamic Change of Template Notation:

If your target platform syntax "collides" with the template notation you might want to change its symbols.

You can change each symbol of the template notation to a new one!

Syntax:

$$\texttt{<<\{[(}\ s_{old}^{(1)} \texttt{ ==> } s_{new}^{(1)},\ s_{old}^{(2)} \texttt{ ==> } s_{new}^{(2)},\ldots,\ s_{old}^{(n)} \texttt{ ==> } s_{new}^{(n)}\ \texttt{)]\}>>}$$

Example:

```
//# Changes brackets so that it‘s easier to distinguish them from XML tags, for example


<<{[(  << ==> ((( , >> ==> )))  )]}>>  //# redefines this bracket << to this bracket (((
                                       //# and >> to )))
```

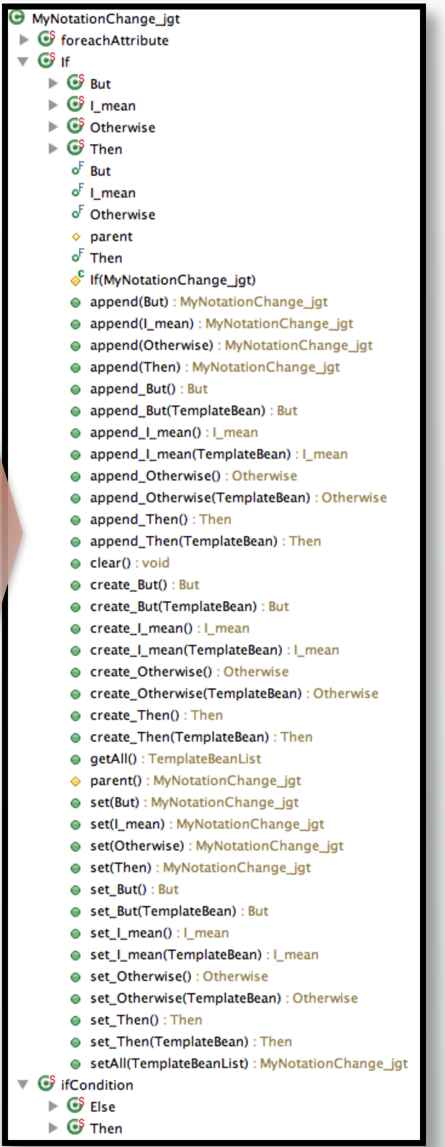
2. Template Notation: Dynamic Notation Change (2)

Larger Example:

Template file: `MyNotationChange.jgt`

```
//# change template notation:

<< {[(  << ==> « , >> ==> »  )]} >>  //# french guillemets

« foreachAttribute --> Attribute.jgt »

«{[(  « ==> [[[ , » ==> ]]]  )]}»  //# re-define again

[[[ If --> Then:      { /* do something here...  */ } |
           Otherwise: { /* otherwise here ... */ } |
           I_mean:    { /* whatever  */ } |
           But:       { /* you can use arbitrary qualifiers */ }
]]]

[[[ {[(  [[[ ==> << , ]]] ==> >>  )]} ]]] // back to orig. notation

<<ifCondition --> Then : { /* then */ } | Else : { /* else */ }>>
```

Automatic Build



```
MyNotationChange_jgt
  ▶ foreachAttribute
  ▼ If
      ▶ But
      ▶ I_mean
      ▶ Otherwise
      ▶ Then
        But
        I_mean
        Otherwise
        parent
        Then
        If(MyNotationChange_jgt)
        append(But) : MyNotationChange_jgt
        append(I_mean) : MyNotationChange_jgt
        append(Otherwise) : MyNotationChange_jgt
        append(Then) : MyNotationChange_jgt
        append_But() : But
        append_But(TemplateBean) : But
        append_I_mean() : I_mean
        append_I_mean(TemplateBean) : I_mean
        append_Otherwise() : Otherwise
        append_Otherwise(TemplateBean) : Otherwise
        append_Then() : Then
        append_Then(TemplateBean) : Then
        clear() : void
        create_But() : But
        create_But(TemplateBean) : But
        create_I_mean() : I_mean
        create_I_mean(TemplateBean) : I_mean
        create_Otherwise() : Otherwise
        create_Otherwise(TemplateBean) : Otherwise
        create_Then() : Then
        create_Then(TemplateBean) : Then
        getAll() : TemplateBeanList
        parent() : MyNotationChange_jgt
        set(But) : MyNotationChange_jgt
        set(I_mean) : MyNotationChange_jgt
        set(Otherwise) : MyNotationChange_jgt
        set(Then) : MyNotationChange_jgt
        set_But() : But
        set_But(TemplateBean) : But
        set_I_mean() : I_mean
        set_I_mean(TemplateBean) : I_mean
        set_Otherwise() : Otherwise
        set_Otherwise(TemplateBean) : Otherwise
        set_Then() : Then
        set_Then(TemplateBean) : Then
        setAll(TemplateBeanList) : MyNotationChange_jgt
  ▼ ifCondition
      ▶ Else
      ▶ Then
```

2. Template Notation: Dynamic Notation Change (3)

Notation Change: *how to use your special notation in each template*

MySpecialNotation.jgt

```
<<{[(  << ==> ((( , >> ==> )))  )]}>>  //# changes tag brackets

((( {[(  //# ==> %//  )]} )))  %// change comment symbol if you
                               %// want to generate LaTeX files
                               %// for example
```

Note: Choose your new symbols wisely! They should be all distinct!

You just have to include your notation at the beginning of your template file:

MyTemplate.jgt

```
<< Include <-- MySpecialNotation.jgt>>   %// change template notation


... %// here comes your normal template
(((SomeVariableText)))
... %// etc.
```

Benefit: *your templates (even with your special notation) become re-usable in other projects because they do not interfere at all with other template notations.*

2. Template Notation: Escape Character

Escape character: *what to do if* << *collides with your target language:*

```
<<VariableName>> //# will be parsed as variable name

\<<NormalText>>  //# << will be parsed as simple text: “<<NormalText>>“
```

A nicer solution is to change the template notation depending on your target platform:

```
<<{[( << ==> $<< )]}>>  //# changes tag start characters

$<<VariableName>>       //# will be parsed as variable name

<<NormalText>>          //# << will be parsed as simple text: “<<NormalText>>“
```

2. Template Notation: Brackets in Inline Templates

Brackets in Inline Templates:

```
public class <<ClassName>>
{
   // some program code here

   <<Implementation -->
   VersionA:
   {
      // first implementation of someMethod()
      public void someMethod(<<DataType>> value)
      {
         process(value);
      }
   }
   |
   VersionB:
   {
      // second implementation of someMethod()
      public void someMethod(<<DataType>> value)
      {
         doSomethingElseWith(value);
      }
   }
}
```

The content of an inline template is opened and closed by { and }

You can use these characters within the content if the structures are symmetrical, i.e. if there are as many open brackets { as closed brackets }

If the structures are not symmetrical you will have to insert \ before the excessive bracket.

In practice, we have never had to use a backslash \ because the structures are usually symmetrical !!!

2. Template Notation: Brackets in Inline Templates (2)

Brackets in Inline Templates:

```
public class <<ClassName>>
{
   <<Implementation --> VersionA:
   {
       // if I use a bracket here, I will
       // have to insert a backslash before \{  it
   }
   |
   VersionB:
   {
       // backslashes are not necessary if the
       // number of opening { and closing }
       // brackets is equal !
   }
}
```

Backslash \ before brackets in inline templates is necessary if the number of opening and closing brackets is not equal!

A backslash \ is not necessary, if the number of opening and closing brackets within an inline template is equal!

2. Template Notation: Renaming Brackets for Templates

Renaming Brackets for Inline Templates:

```
public class <<ClassName>>
{
  ... // some stuff
 <<{[(  { ==> {( , } ==> )}  )]}>> // change brackets

 <<Implementation -->
   VersionA:
   {(
       // if I use my special brackets,
       // I don‘t have to use a backslash before {
   )}
   |
   VersionB:
   {(
        // backslashes are not necessary if
        // my special brackets are not used
        // within the inline template.
   )}
}
```

How to avoid backslashes!

You can also change the brackets to any symbol you want, so that they cannot interfere with any bracket within the program code of the inline template.

2. Template Notation: Mixing Sub Template Types

Mixing normal sub templates and inline sub templates:

```
<<foreachElement --> First.jgt | Inside: { /* the inline template */ } | Third.jgt>>
```

Filling data into the template:

```
template.foreachElement.append_Third_jgt();    // the order and the number is arbitrary
template.foreachElement.append_Inside();
template.foreachElement.append_Inside();
template.foreachElement.append_First_jgt();
template.foreachElement.append_Inside();
```

2. Template Notation: Arbitrary Sub Templates

Arbitrary sub templates:

```
<<foreachElement --> ... >>  //# an arbitrary TemplateBean can be
                             //# added;
                             //# there will be no type checking
                             //# not at compile-time and not at run-time
```

No type safety!

You can add any arbitrary TemplateBean instance via following command:

```
template.foreachElement.append(new ArbitraryBean_jgt());
```

Sometimes it is necessary to have this flexibility!

2. Template Notation: Text Formatting Operators

Text Formatting Operators:

```
//#  text operator: /

<<VariableName/>>   //# Has only an effect if VariableName consists of more than one line:
                    //# The / before >>  deletes all previous white space lines
                    //# including the one with the text operators

<<foreachElement --> SubTemplate.jgt />>  //#   the slash / means, that the line of the
                                          //#   text operator will be deleted
                                          //#   if it contains only whitespaces
                                          //#   and also all previous whitespace lines, i.e
                                          //#   this operator basically deletes all excessive
                                          //#   after the last inserted subtemplate.

//#  flowtext operator: ~~

<< ~~ VariableName >>                     //#   inserts the text of the variable
                                          //#   as flowtext if it exceeds more than one line
                                          //#   (and not as left-aligned blocktext
                                          //#   which is the default).

<< ~~ foreachElement --> SubTemplate.jgt >>  //#   inserts the text which consists of all
                                             //#   subtemplates as flowtext.
```

2. Template Notation: Value Propagation

Template Bean Value Propagation: *values are propagated by default*

Template file: `Template.jgt`

```
<<ClassName>> clazz = new <<ClassName>>();

<< foreachElement --> Inline:
{
   // do something with <<ClassName>>
}
|
SubTemplate.jgt
>>
```

Template file: `SubTemplate.jgt`

```
// do something else with <<ClassName>>

<<ClassName>>
```

Filling the template beans:

```
template.setClassName("MyClassName");
template.foreachElement.append_Inline();
template.foreachElement.append_SubTemplate_jgt();
```

You do not have to set the value of <<ClassName>> *in the sub templates, because it will be propagated from the calling template,* ***if*** *you don‘t set it in the sub template*

Checking/Setting value propagation:

```
template.isValuePropagationOn(); // default is On
template.turnValuePropagationOn(); // switch it On
template.turnValuePropagationOff(); // switch it Off
template.setValuePropagation(booleanValue); // set it
```

3. Updating Source Files

Updating Source Files

3. Updating Source Files: Generator

Generator:

```java
public class MyGenerator extends AbstractGenerator
{
   .
   . // Constructor stuff
   .
    @Override
    public void generate()
    {
       .
       . // filling the template bean with data
       .

       String path     = template.getPackageName().replace('.', '/');
       String filepath = path + "/" + template.getClassName() + ".java";
       updateSourceFile ( filepath ,   template.toString() ) ;
    }
}
```

Updates existing source files with new source text.

Does not overwrite so-called Protected Regions

3. Updating Source Files: Protected Regions

Protected Regions:

```
// {{ProtectedRegionStart::NameOfTheRegion}}

    public WhateverDataType computeSomethingSpecial()
    {
        ... // some stuff
    }

// {{ProtectedRegionEnd}}
```

This text in between here is protected from being overwritten!

Protected region tags just have to be ***the only control element in one line*** *and preferably after the comment symbol of the target platform:*

In case of C/C++ : `/* {{ProtectedRegionStart::NameOfTheRegion}} */`

In case of Java/C++/C# : `// {{ProtectedRegionStart::NameOfTheRegion}}`

In Linux Shell scripts : `#  {{ProtectedRegionStart::NameOfTheRegion}}`

The same holds for the protected region end tag: `// {{ProtectedRegionEnd}}`

3. Updating Source Files: Protected Regions (2)

Example:

```
package <<PackageName>>;

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

    << foreachAttribute --> GetterSetter.jgt />>

    // {{ProtectedRegionStart::SpecialMethods}}
    public void someSpecialMethodAsExample()
    {
        // do something
    }
    // {{ProtectedRegionEnd}}
}
```

- *Protected Regions only have an effect when updating existing source files.*
- *They* ***do not*** *affect the code generation process with template beans.*
- *They* ***do not*** *belong to the Template Notation but instead to the Text Region Notation.*
- *The Text Region Notation consists only of the syntax for Protected Regions.*

4. Configuration

Configuration

4. Configuration: Overview

There are just two tasks to configure:

A. Configure automatic build (IDE) for template bean creation.

B. Configure the code generation process for your project.

4.A. Configuration: Template Beans

Configure the creation of template beans:

1. Read the README file of the distribution.
2. Adjust `jiowa.codegen.properties` (if you have to)
3. Write your templates in the directory specified in `jiowa.codegen.properties`.
4. Tell your IDE to automatically rebuild if any of the templates or properties change (pre-configured for Eclipse).

That‘s it! ✔

jiowa-codegen distribution *is a Maven project (optimized for Eclipse) !*

4.B. Configuration: Code Generation

Configure Your Code Generation Process:

1. Configure the code generator output directory in `jiowa.codegen.properties`.
2. Write several code generators for your project.
3. Register them at the `JiowaCodeGeneratorEngine` and call the `start()`-Method.

That‘s it! ✔

jiowa-codegen distribution is a Maven project (optimized for Eclipse) !

4. Configuration: Property File

Configuration of the property file: `jiowa.codegen.properties`

```
# CodeGen properties:
jiowa.codegen.generator.output.directory=src/gen/java

# TemplateBean properties:
# comma separated list of template directories
jiowa.template.directories=src/main/resources/generator/templates
# template file suffix
jiowa.template.file.suffix=.jgt

# Definitions for template bean generator:
# output directory
jiowa.template.beangen.output.directory=src/gen-beans/java
# package name for template beans
jiowa.template.beangen.output.package=com.jiowa.template.bean
```

The red properties are the only ones you might want to change !

Everything else is pre-configured so that it just works!

4. Configuration: Template Files

Some hints on template files:

1. Text file encoding: UTF-8 is recommended.
2. Tabulators will be replaced automatically by 4 spaces in memory. IDE should be configured to use spaces instead of tabulators.
3. More than one template directory at once can be used (comma-separated list) in `jiowa.codegen.properties`
4. All templates of the template directories are in the same namespace, i.e. they can be stored and moved in(to) arbitrary sub-directories without the need to change any template or generator.
5. Tell your IDE to use syntax highlighting of your target platform (Java, C++, ...) for template files (`.jgt` suffix). This is a workspace feature which cannot be pre-configured in the distributed project.

jiowa-codegen distribution *is a Maven project (optimized for Eclipse) !*

4. Configuration: Configuration Class

Configuration Class supports Property Overloading:

```
JiowaCodeGenConfig config =
                      new JiowaCodeGenConfig(“my.codegen.properties“, “jiowa.codegen.properties");
```

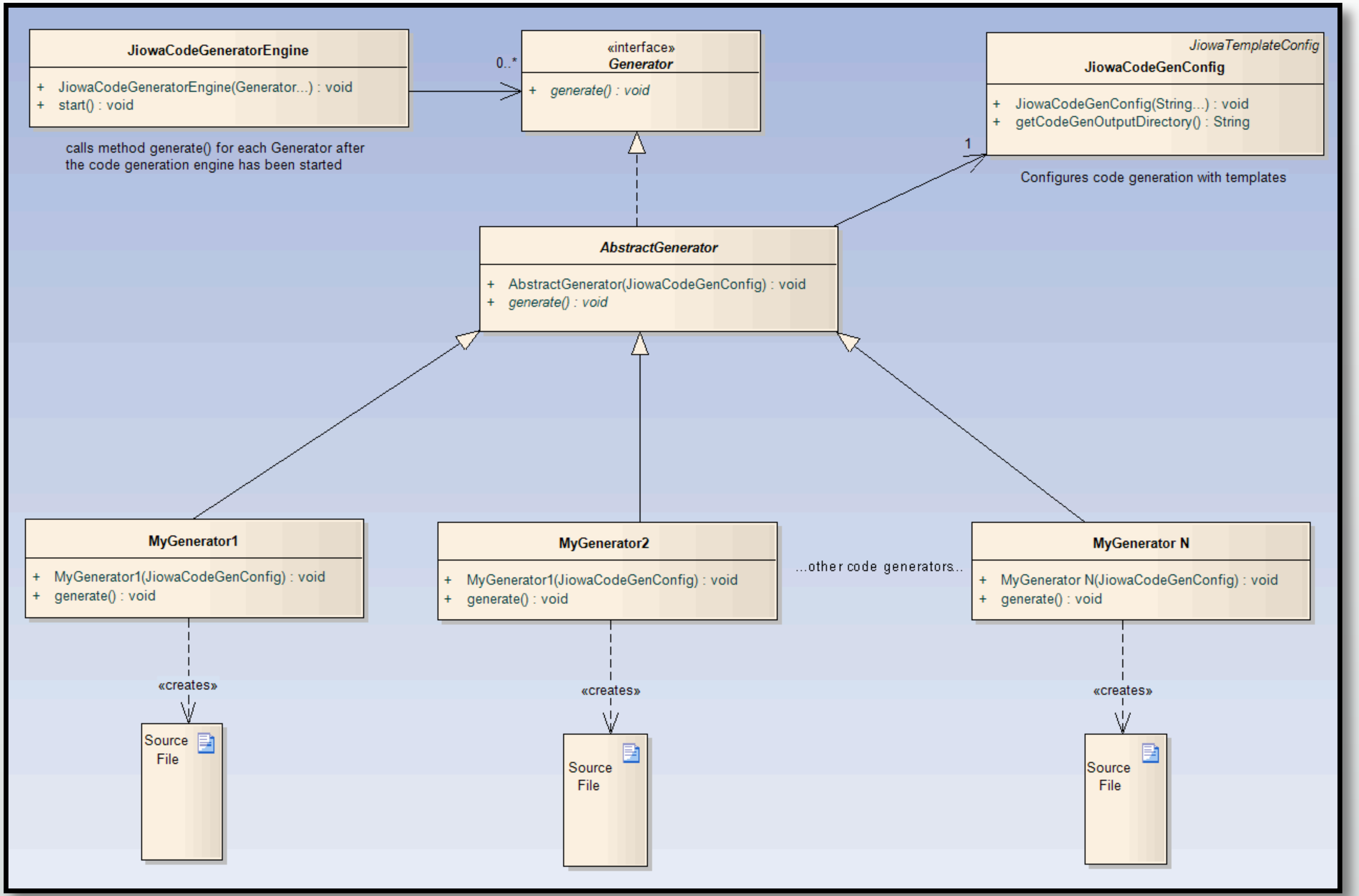
The values in “my.codegen.properties“ override the values in “jiowa.codegen.properties“.

4. Configuration: Generator Engine

Generator Engine: registers all your generators and starts them at once

```
// Initialize configuration class:
JiowaCodeGenConfig config = new JiowaCodeGenConfig("jiowa.codegen.properties");

// Create your generators:
Generator myGenerator1 = new MyGenerator1(config);
Generator myGenerator2 = new MyGenerator2(config);
Generator myGenerator3 = new MyGenerator3(config);


// Create instance of code generator engine:
// It calls myGenerator?.generate() for each registered Generator.
JiowaCodeGeneratorEngine engine = new JiowaCodeGeneratorEngine(myGenerator1,
                                                                myGenerator2,
                                                                  myGenerator3);

// Start the code generation process:
engine.start();
```

4. Configuration: Generator Engine Overview


JiowaCodeGeneratorEngine
+ JiowaCodeGeneratorEngine(Generator...) : void
+ start() : void
calls method generate() for each Generator after the code generation engine has been started
0..*
«interface»
Generator
+ generate() : void
JiowaTemplateConfig
JiowaCodeGenConfig
+ JiowaCodeGenConfig(String...) : void
+ getCodeGenOutputDirectory() : String
1
Configures code generation with templates
AbstractGenerator
+ AbstractGenerator(JiowaCodeGenConfig) : void
+ generate() : void
MyGenerator1
+ MyGenerator1(JiowaCodeGenConfig) : void
+ generate() : void
MyGenerator2
+ MyGenerator1(JiowaCodeGenConfig) : void
+ generate() : void
...other code generators...
MyGenerator N
+ MyGenerator N(JiowaCodeGenConfig) : void
+ generate() : void
«creates»
«creates»
«creates»
Source File
Source File
Source File

5. Distribution

Distribution

5. Distribution

Distribution of **Jiowa CodeGen:**

- Consists of a pre-configured Maven project (optimized for Eclipse):
  - Distribution: `jiowa-codegen_2.1_example_mvn.zip`
  - Contains a code generation example which can be taken as prototype for your own project.
  - Everything has been packaged into the zip file: this tutorial, the code examples, ... etc.

You can just import it into your IDE as Maven project!

5. Distribution: Future Release of Jiowa MDSD

Outlook on **Jiowa-MDSD:** *(to be published soon...)*

- `jiowa-mdsd = { jiowa-codegen , jiowa-uml }`
- Supports all models in UML/XMI-2.x - Format
- Enterprise Architect, Visual Paradigm, IBM RSA, MagicDraw, Papyrus, EclipseUML-Plugin, etc.
- Supports also UML Diagrams of any modeling tool despite there is no OMG specification on UML Diagrams for UML/XMI-2.x so far (the storage format for diagrams is very different between the tools)

MDSD = Model Driven Software Development

6. Why using it?

Why using it?

6. Why using it?

Why using it?

- It‘s easy, efficient, and it just works!
- Existing features are enough! Anything else you can do within Java yourself. No need to invent more template notation features.
- Syntax highlighting for the target language and not for the template language.
- It‘s an open framework! It does not force you to do things in a specific way. You can use it in the way it's prepared, but you can also use it your own way within Java.
- Minimized framework! Only as many features as necessary. By design, it avoids unnecessary complex features and/or language constructs. They are just not needed!
- This tutorial already explained all features of this framework to you! No need for a 300+ pages handbook and weeks of training and consulting! Just start using it!

If you cannot explain it simply,
you don‘t understand it well enough!
- Albert Einstein

7. License

License

7. License

JIOWA Code Generation Framework is licensed under the

JIOWA Backlinking License v1.0:

- Free for commercial and non-commercial use ...
  - **if** you set a public link from your website(s) to www.jiowa.de stating that you use the JIOWA Code Generation Framework (if you don't own a website you can also set a backlink from your favourite social media account),
  - **and** you leave all our copyright/license notices inside the distribution.
  - Example link text:
    We are using the JIOWA Code Generation Framework!
- That's all! Don't you think this is a fair agreement?
- Full license text: www.jiowa.de/license.html
- **Get it here...** www.jiowa.de/download.html Have fun! :-)

JIOWA Code Generation Framework

Questions ?

Feedback ?

codegen@jiowa.de

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

Slideshare: www.slideshare.net/Robert_Mencl/jiowa-code-generator-framework/