

JIOWA Code Generation Framework

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

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

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

JIOWA Code Generation Framework

- 1. Simple Examples**
- 2. Template Notation**
- 3. Updating Source Files**
- 4. Configuration**
- 5. Distribution**
- 6. Why Using It?**
- 7. License**

I. Simple Examples

Email Template

I. 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
<< ... >>

I. 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!

I. Simple Examples: Email Template (3)

sub template 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 sub template indicator and `|` is the or-symbol
 for different possible sub templates.

I. 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>>
```

Automatic
Build

TemplateBean Class: Letter_jgt

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

compile-time safety
for template syntax

I. Simple Examples: Email Template (5)

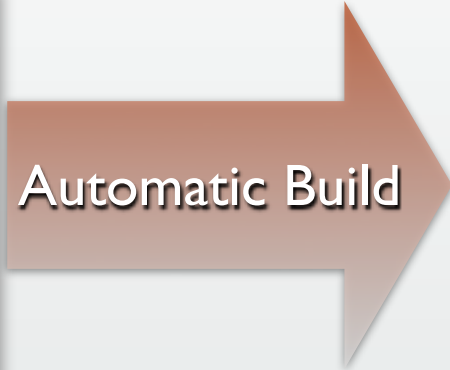
Template file: Letter.jgt

```
Dear <<Salutation -->
    Mr: {Mr.} | Mrs: {Mrs.}>> <<Name>>,
we hereby want to inform you...
bla bla bla .
Best regards,
<<ContactPerson>>
```

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);
```

TemplateBean Class:
Letter_jgt



```
Letter_jgt
├─ Letter_jgt()
├─ Letter_jgt(TemplateBean)
├─ id(): String
├─ getName(): String
├─ setName(String): Letter_jgt
├─ getContactPerson(): String
├─ setContactPerson(String): Letter_jgt
├─ Salutation: Salutation
├─ Salutation
├─ Mr
│   ├── Mr()
│   ├── Mr(TemplateBean)
│   └─ id(): String
│       └─ Letter_jgt(): Letter_jgt
├─ Mrs
├─ getAll(): TemplateBeanList
├─ setAll(TemplateBeanList): Letter_jgt
├─ setSubTemplate(String): TemplateBean
├─ setSubTemplate(String, TemplateBean): TemplateBean
├─ clear(): void
├─ prepend(Mr): Letter_jgt
├─ prepend_Mr(): Mr
├─ prepend_Mr(TemplateBean): Mr
├─ append(Mr): Letter_jgt
├─ append_Mr(): Mr
├─ append_Mr(TemplateBean): Mr
├─ set(Mr): Letter_jgt
├─ set_Mr(): Mr
├─ set_Mr(TemplateBean): Mr
├─ prepend(Mrs): Letter_jgt
├─ prepend_Mrs(): Mrs
├─ prepend_Mrs(TemplateBean): Mrs
├─ append(Mrs): Letter_jgt
├─ append_Mrs(): Mrs
├─ append_Mrs(TemplateBean): Mrs
├─ set(Mrs): Letter_jgt
├─ set_Mrs(): Mrs
└─ set_Mrs(TemplateBean): Mrs
```


I. 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:

```
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
```

I. Simple Examples

Java Class Template

I. 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`

I. 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:

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.

```
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.

I. Simple Examples: Java Class Template (3)

Template file: Class.jgt

```
package <<PackageName>>;

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

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

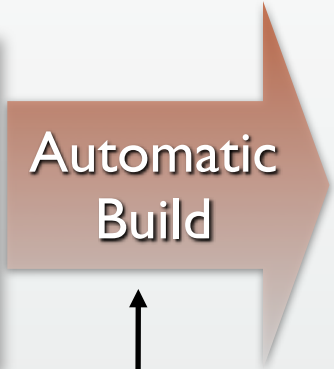
Template file: Attribute.jgt

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

Template file: GetterSetter.jgt

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

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



compile-time safety for template syntax

TemplateBean Class: Class_jgt

```
Class_jgt
├─ Class_jgt()
├─ Class_jgt(TemplateBean)
├─ id() : String
├─ getPackageName() : String
├─ setPackageName(String) : Class_jgt
├─ getClassName() : String
├─ setClassName(String) : Class_jgt
├─ F foreachAttribute : foreachAttribute
└─ S foreachAttribute
    ├── getAll() : TemplateBeanList
    ├── setAll(TemplateBeanList) : Class_jgt
    ├── setSubTemplate(String) : TemplateBean
    ├── setSubTemplate(String, TemplateBean) : TemplateBean
    ├── clear() : void
    ├── prepend(Attribute_jgt) : Class_jgt
    ├── prepend_Attribute_jgt() : Attribute_jgt
    ├── prepend_Attribute_jgt(TemplateBean) : Attribute_jgt
    ├── 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
    ├── prepend(GetterSetter_jgt) : Class_jgt
    ├── prepend_GetterSetter_jgt() : GetterSetter_jgt
    ├── prepend_GetterSetter_jgt(TemplateBean) : GetterSetter_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
```

I. 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)
  id() : String
  getDataType() : String
  setDataType(String) : Attribute_jgt
  getAttributeName() : String
  setAttributeName(String) : Attribute_jgt
```

I. Simple Examples: Java Class Example (5)

Class.jgt

```
package <<PackageName>>;

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

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

Attribute.jgt

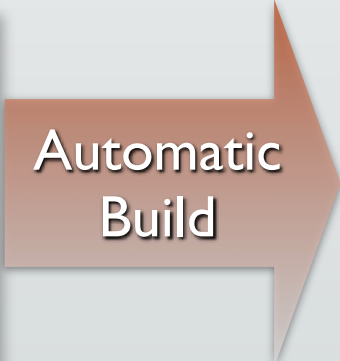
```
protected <<DataType>> <<AttributeName>>;
```

GetterSetter.jgt

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

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

TemplateBean Class: GetterSetter_jgt



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

I. Simple 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 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!

I. Simple Examples: Java Class Example (7)

Graphical Model:

- UML:
 - Eclipse-based UML/EMF: use Java UML API from the eclipse project.
 - UML formats of other modeling tools: use Java UML API of the specific vendor if there is one.
- Any other graphical notation:
 - you will just need a Java API to read data from your model instance.

I. Simple Examples: Java Class Example (8)

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.*

I. Simple Examples: Java Class Example (9)

Example: Iterating over existing classes and generating a Java bean for each one of them.

Instead of a simple Java bean we also could generate some technology layer classes in a real world application!

```

public class MyGenerator extends AbstractGenerator
{
    .
    .
    .
    @Override
    public void generate()
    {
        .
        .
        .
        for (Class<?> clazz : someExampleListOfClasses)
        {
            Class_jgt t = new Class_jgt().setPackageName("example").setClassName(clazz.getSimpleName() + "Bean");

            for (Field field : clazz.getDeclaredFields())
            {
                Attribute_jgt attr = t.foreachAttribute.append_Attribute_jgt().
                    setDataType(field.getType().getSimpleName()).
                    setAttributeName(field.getName());

                t.foreachAttribute.append_GetterSetter_jgt(attr);
            }

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

```

Construct
name of
Java bean

copy
constructor
for attributes



I. Simple Examples: Java Class by using Inline Templates

Java Class with Inline Templates

I. Simple Examples: Java Class by using Inline Templates

Template file: MyClass.jgt

TemplateBean Class: MyClass_jgt

```

package <<PackageName>>;

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

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

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



Automatic Build

Inline Templates are represented by nested classes within the TemplateBean

```

MyClass_jgt
├── MyClass_jgt()
├── MyClass_jgt(TemplateBean)
├── id(): String
├── getPackageName(): String
├── setPackageName(String): MyClass_jgt
├── getClassName(): String
├── setClassName(String): MyClass_jgt
├── foreachAttribute: foreachAttribute
├── foreachAttribute
├── Attr
│   ├── Attr()
│   ├── Attr(TemplateBean)
│   ├── id(): String
│   ├── MyClass_jgt: MyClass_jgt
│   ├── getDataType(): String
│   ├── setDataType(String): Attr
│   ├── getAttributeName(): String
│   └── setAttributeName(String): Attr
├── GetterSetter
│   ├── GetterSetter()
│   ├── GetterSetter(TemplateBean)
│   ├── id(): String
│   ├── MyClass_jgt: MyClass_jgt
│   ├── getDataType(): String
│   ├── setDataType(String): GetterSetter
│   ├── getAttributeName(): String
│   └── setAttributeName(String): GetterSetter
├── getAll(): TemplateBeanList
├── setAll(TemplateBeanList): MyClass_jgt
├── setSubTemplate(String): TemplateBean
├── setSubTemplate(String, TemplateBean): TemplateBean
├── clear(): void
├── prepend(Attr): MyClass_jgt
├── prepend_Attr(): Attr
├── prepend_Attr(TemplateBean): Attr
├── append(Attr): MyClass_jgt
├── append_Attr(): Attr
├── append_Attr(TemplateBean): Attr
├── set(Attr): MyClass_jgt
├── set_Attr(): Attr
├── set_Attr(TemplateBean): Attr
├── prepend(GetterSetter): MyClass_jgt
├── prepend_GetterSetter(): GetterSetter
├── prepend_GetterSetter(TemplateBean): GetterSetter
├── append(GetterSetter): MyClass_jgt
├── append_GetterSetter(): GetterSetter
├── append_GetterSetter(TemplateBean): GetterSetter
├── set(GetterSetter): MyClass_jgt
├── set_GetterSetter(): GetterSetter
├── set_GetterSetter(TemplateBean): GetterSetter
    
```

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

Example: Iterating over existing classes and generating a Java bean for each one of them.

Instead of a simple Java bean we also could generate some technology layer classes in a real world application!

```
public class MyGenerator extends AbstractGenerator
{
    .
    .
    .
    @Override
    public void generate()
    {
        .
        .
        .
        for (Class<?> clazz : someExampleListOfClasses)
        {
            MyClass_jgt t = new MyClass_jgt().setPackageName("example").setClassName(clazz.getSimpleName()+"Bean");

            for (Field field : clazz.getDeclaredFields())
            {
                Attr attr = t.foreachAttribute.append_Attr().
                    setDataType(field.getType().getSimpleName()).
                    setAttributeName(field.getName());

                t.foreachAttribute.append_GetterSetter(attr);
            }

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

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 sub template structures:


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

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

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

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:

Template file: Template.jgt

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

Template file: SubTemplate.jgt

```
// ... some template content in here ...
```

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 t = new Template_jgt();
t.foreachElement.prepend_SubTemplate_jgt();
t.foreachElement.append(new SubTemplate_jgt()); // possible
t.foreachElement.append_SubTemplate_jgt();      // possible as well
// inserts exactly one element into the list and removes everything else
t.foreachElement.set_SubTemplate_jgt();
System.out.println(t); // template output
```

... see in [JavaDoc of generated TemplateBeans](#) for many other methods

to set, prepend & append data !

2. Template Notation: Sub Templates (2)

Language independence of sub structure identifiers:

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

sub structure identifier

German:

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

French:

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

Free Text:

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

```

IdentifierTest_jgt
├── IdentifierTest_jgt()
├── IdentifierTest_jgt(TemplateBean)
├── id() : String
├── F pourChaqueElement : pourChaqueElement
├── S pourChaqueElement
├── F fuerJedesElement : fuerJedesElement
├── S fuerJedesElement
├── F Whatever_you_like : Whatever_you_like
├── S Whatever_you_like
├── getAll() : TemplateBeanList
├── setAll(TemplateBeanList) : IdentifierTest_jgt
├── setSubTemplate(String) : TemplateBean
├── setSubTemplate(String, TemplateBean) : TemplateBean
├── clear() : void
├── prepend(SubTemplate_jgt) : IdentifierTest_jgt
├── prepend_SubTemplate_jgt() : SubTemplate_jgt
├── prepend_SubTemplate_jgt(TemplateBean) : SubTemplate_jgt
├── append(SubTemplate_jgt) : IdentifierTest_jgt
├── append_SubTemplate_jgt() : SubTemplate_jgt
├── append_SubTemplate_jgt(TemplateBean) : SubTemplate_jgt
├── set(SubTemplate_jgt) : IdentifierTest_jgt
├── set_SubTemplate_jgt() : SubTemplate_jgt
├── set_SubTemplate_jgt(TemplateBean) : SubTemplate_jgt

```

2. Template Notation: Sub Templates (3)

Choosing between different sub templates:

Template file: Template.jgt

or-symbols

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

In your generator code:

```
Template t = new Template_jgt();

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

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

Type Safety!

Only instances of First_jgt, Second_jgt and Third_jgt are processed!

2. Template Notation: Inline Templates

Inline Templates:

Template file: Template.jgt

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

qualifier symbol :

point of text insertion

inline template name **Extends**

inline template text between { ... }

Filling Data into the Template Bean:

```
Template t = new Template_jgt();

// set ClassName:
t.setClassName("MyClass");
// set SuperClassName:
t.ifHasSuperClass.set_Extends().setSuperClassName("MySuperClass"); // 1st possibility
// or
t.ifHasSuperClass.set(new Extends().setSuperClassName("MySuperClass")); // 2nd possibility

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


2. Template Notation: Inline Templates (2)

Switching between sub templates:

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

/, text operator deletes the last comma which remains after the last interface name

Filling values into this template bean:

```
Template t = new Template_jgt();

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

System.out.println(t); // template output
```

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>> } /,>>
{
    ... // some stuff
}
```



Simplify

```
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>>, } /,>>
```

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:
leftmost non-whitespace character position defines offset column from which all subsequent lines are adjusted.

Important if you want to generate visually appealing code !

plus one return character

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 line feeds (LF) as there are rows.

just empty text

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

Filling data into the template bean:

```
Template t = new Template_jgt();

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

System.out.println(t); // template output
```

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 (2)

Larger Example:

Template file: `NotationChangeExample.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 */ }>>
    
```



```

NotationChangeExample.jgt
  C NotationChangeExample.jgt()
  C NotationChangeExample.jgt(TemplateBean)
  C id() : String
  F foreachAttribute : foreachAttribute
  S foreachAttribute
  F ifCondition : ifCondition
  S ifCondition
  F If : If
  S If
  S But
  S L_mean
  S Otherwise
  S Then
  C getAll() : TemplateBeanList
  C setAll(TemplateBeanList) : NotationChangeExample.jgt
  C setSubTemplate(String) : TemplateBean
  C setSubTemplate(String, TemplateBean) : TemplateBean
  C clear() : void
  C prepend(But) : NotationChangeExample.jgt
  C prepend_But() : But
  C prepend_But(TemplateBean) : But
  C append(But) : NotationChangeExample.jgt
  C append_But() : But
  C append_But(TemplateBean) : But
  C set(But) : NotationChangeExample.jgt
  C set_But() : But
  C set_But(TemplateBean) : But
  C prepend(L_mean) : NotationChangeExample.jgt
  C prepend_L_mean() : L_mean
  C prepend_L_mean(TemplateBean) : L_mean
  C append(L_mean) : NotationChangeExample.jgt
  C append_L_mean() : L_mean
  C append_L_mean(TemplateBean) : L_mean
  C set(L_mean) : NotationChangeExample.jgt
  C set_L_mean() : L_mean
  C set_L_mean(TemplateBean) : L_mean
  C prepend(Otherwise) : NotationChangeExample.jgt
  C prepend_Otherwise() : Otherwise
  C prepend_Otherwise(TemplateBean) : Otherwise
  C append(Otherwise) : NotationChangeExample.jgt
  C append_Otherwise() : Otherwise
  C append_Otherwise(TemplateBean) : Otherwise
  C set(Otherwise) : NotationChangeExample.jgt
  C set_Otherwise() : Otherwise
  C set_Otherwise(TemplateBean) : Otherwise
  C prepend(Then) : NotationChangeExample.jgt
  C prepend_Then() : Then
  C prepend_Then(TemplateBean) : Then
  C append(Then) : NotationChangeExample.jgt
  C append_Then() : Then
  C append_Then(TemplateBean) : Then
  C set(Then) : NotationChangeExample.jgt
  C set_Then() : Then
  C set_Then(TemplateBean) : Then
    
```

2. Template Notation: Dynamic Notation Change (3)

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

MyTeXNotation.jgt

```
<< {[ ( // # ==> %// ) ]} >> // changes comment symbol to
// something more suitable for TeX
```

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

TeXTemplate.jgt

```
<< {[ ( MyTeXNotation.jgt ) ]} >> // imports template notation
// change from file
// MyTeXNotation.jgt

... // here comes your normal LaTeX template
<<SomeVariableText>>
... // etc.
```

You just have to put or include your notation at the beginning of your template file.

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
    // numbers of opening { and closing }
    // brackets are equal !
  }
}
```

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

A backslash \ is not necessary, if the numbers of opening and closing brackets within an inline template are 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:

Template file: Template.jgt

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

Filling data into the template:

```
Template t = new Template_jgt();  
  
t.foreachElement.append_Third_jgt(); // the order and the number is arbitrary  
t.foreachElement.append_Inside();  
t.foreachElement.append_Inside();  
t.foreachElement.append_First_jgt();  
t.foreachElement.append_Inside();  
  
System.out.println(t); // template output
```


2. Template Notation: Value Propagation

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>>
```

*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.*

Using the template bean:

```
Template_jgt t = new Template_jgt();

t.setClassName("MyClass");
t.foreachElement.append_Inline();
t.foreachElement.append_SubTemplate_jgt();

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

Output

```
MyClass clazz = new MyClass();

// do something with MyClass
// do something else with MyClass
```

Value propagation also works for sub template instances!

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

3. Updating Source Files

Updating Source Files

3. Updating Source Files: Generator

Generator:

```
public class MyGenerator extends AbstractGenerator
{
    . // Constructor stuff
    .
    @Override
    public void generate()
    {
        Template_jgt t = new Template_jgt();
        . // filling the template bean with data
        .

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

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`

```
# -----  
# Only for code generation:  
# -----  
jiowa.codegen.generator.output.directory=src/gen/java  
  
# -----  
# Only for template bean generation:  
# -----  
# comma separated list of template directories  
jiowa.template.directories=src/main/resources/generator/templates  
# template file suffix: comma separated list of file suffixes  
jiowa.template.file.suffix=.jgt  
# package name for template beans  
jiowa.template.beangen.output.package=com.jiowa.template.bean  
# template bean output directory: should not be changed!  
jiowa.template.beangen.output.directory=src/gen-beans/java
```

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

Please keep in mind, that `jiowa.codegen.properties` and the source directories are also used in Maven's `pom.xml` \Rightarrow If you change them here, you will also have to change them in `pom.xml`.

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

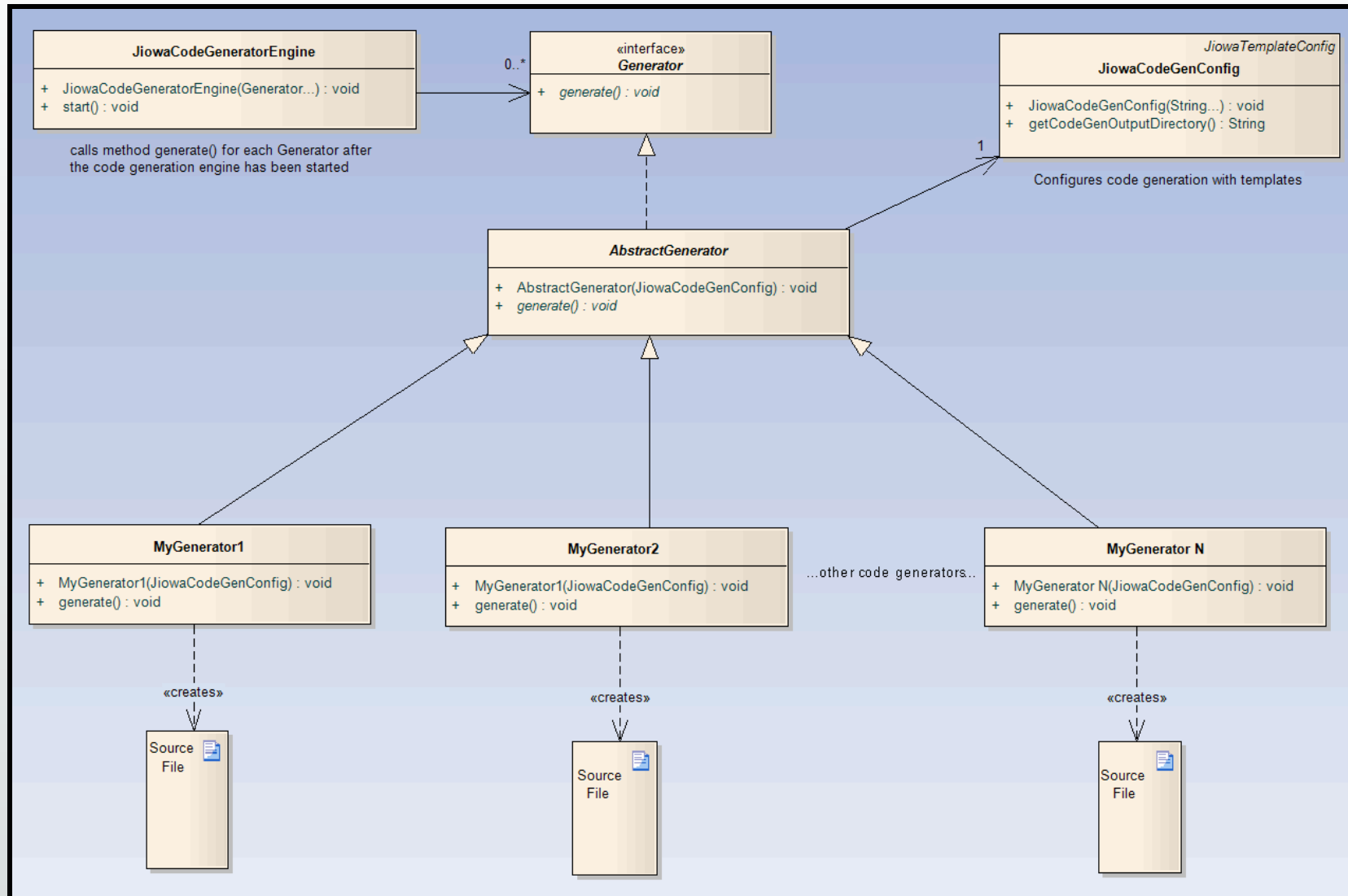
```
// Initialize configuration class:
JiowaCodeGenConfig config = new JiowaCodeGenConfig("jiowa.codegen.properties");

// 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



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!

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.
- 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/

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