# pxDoc

# Reference Guide

## pxDoc V1.1

Last document update: Sept-20th 2018

© EXTREME MODELING SOFTWARE 2018

Table of Contents

1. pxDoc Overview ..... 6
1. Overview ..... 6
1. A pragmatic and powerful document generation framework ..... 6
2. What pxDoc is not ..... 6
2. Main concepts ..... 7
3. Reusable pxDocGenerators ..... 8
4. Stylesheet-Dependent pxDocGenerators ..... 10
5. pxDoc Tooling Features ..... 11
1. Integrate your pxDoc projects with pxGen ..... 11
2. Content assist ..... 11
3. pxDoc Validation & Quickfixes ..... 12
4. Java Integration ..... 13
5. Automatic Formatting ..... 13
6. Include external (documentation or not) dialects ..... 13
6. Mix Manual and Generated document parts ..... 13
2. Create, publish and launch projects ..... 15
1. Project creation ..... 15
2. Launching the generation ..... 20
3. pxGen ..... 21
1. Fine-tune the Eclipse UI ..... 22
2. Generate Eclipse Feature and Repository to publish the Generator ..... 22
3. Template concepts ..... 24
1. pxDocGenerators and pxDocModules ..... 24
2. Keyword 'template' ..... 26
3. Keyword 'apply' ..... 27
4. Usages of template and apply keywords ..... 27
5. Keyword 'function' ..... 30
6. Implicit 'it' parameter ..... 31
7. Xbaselib extensions ..... 31
8. var / val ..... 32
4. Document concepts ..... 34



© EXTREME MODELING SOFTWARE 2018

1. Documents and Subdocuments ..... 34
1. Keyword 'document' ..... 34
2. About Sub-Documents and manual parts ..... 36
3. Keyword 'subDocument' ..... 36
2. Import content from other languages ..... 37
1. Keyword 'language' ..... 37
3. Applying styles ..... 38
1. Get Styles from a stylesheet ..... 39
2. Use variable styles ..... 40
3. Keyword 'styleApplication' ..... 41
4. Keyword 'HeadingN' ..... 42
4. Modifying styles ..... 43
1. Keyword 'bold' ..... 43
2. Keyword 'italic' ..... 43
3. Keyword 'underline' ..... 44
4. Keyword 'strike' ..... 44
5. Keyword 'subscript' ..... 44
6. Keyword 'superscript' ..... 44
7. Keyword 'color' ..... 44
8. Keyword 'font' ..... 45
9. Usages ..... 46
5. Pages and line breaks, Sections ..... 47
1. Keyword 'paragraphBreak' ..... 47
2. Keyword 'newLine' ..... 47
3. Keyword 'nextColumn' ..... 47
4. Keyword 'newPage' ..... 47
5. Keyword 'section' ..... 47
6. Breaks usages ..... 48
7. Keyword 'samePage' ..... 50
6. Include images ..... 51
1. Keyword 'image' ..... 51
2. Usages of image ..... 52

7. Include bookmarks and hyperlinks ..... 54
1. Keyword 'bookmark' ..... 54
2. Keyword 'hyperlink' ..... 54
8. Fields and Table of Content ..... 55
1. Keyword 'toc' ..... 55
2. Keyword 'field' ..... 56
9. Tables ..... 57
1. Principles ..... 57
2. Keyword 'table' ..... 59
3. Keyword 'columnProperties' ..... 60
4. Keyword 'row' ..... 62
5. Keyword 'cell' ..... 62
6. Usages ..... 63
10. About Measures ..... 65
5. pxDoc extensibility ..... 66
1. Common extensibility tasks on the UI project ..... 66
1. Adapt the generation launching wizard ..... 66
2. Bind variable styles before deploying a generator ..... 72
2. pxDoc - Common Libraries and Plugins ..... 73
1. Importing sources into your workspace ..... 74
2. Common Services / *org.pragmaticmodeling.pxdoc.common.lib* ..... 74
3. EMF Services / *org.pragmaticmodeling.pxdoc.emf.lib* ..... 76
4. Diagram Services / *org.pragmaticmodeling.pxdoc.diagrams.lib* ..... 78
5. Images Services / *org.pragmaticmodeling.pxdoc.images.lib* ..... 81
6. Install pxDoc License ..... 82
1. Evaluation License ..... 82
2. Personal License ..... 84
3. Fixed License ..... 85
4. Floating License ..... 87
1. Install Floating License Server and get its Hardware ID ..... 87
2. Specify Floating License Server location in pxDoc ..... 87



© EXTREME MODELING SOFTWARE 2018

# 1. pxDoc Overview

## 1. Overview

### 1. A pragmatic and powerful document generation framework

pxDoc is designed to:

- mix manual and generated document fragments, as it is rarely possible to automatically generate 100% of a document,
- support the Microsoft Word target (and other open xml formats in the near future),
- fully support of your specific Word templates,
- easily mix document keywords with Java instructions with a dedicated development language and editor,
- ability to get any Java object as input (no need to learn complex and limited query systems to get in touch with your data),
- ability to organize documentation templates into reusable modules,
- ability to easily implement generation algorithms of arbitrary complexity,
- ability to develop document generators independently of the Word templates, with variable styles,
- ability to generate complex tables with vertical merge,
- ability to debug document generators (breakpoints, step-by-step execution...),
- ability to generate document of arbitrary complexity and volume.

pxDoc does not require MS Word nor other third party libraries to generate documents: e.g. documents can be generated on Linux operating systems, etc.

### 2. What pxDoc is not

No, you won't design your document generators directly in MS Word with pxDoc. Nor will you have to struggle with a complex and raw java API...

Both of these common document generation approaches have severe limitations and/or drawbacks.



© EXTREME MODELING SOFTWARE 2018

| | pxDoc | Wysiwyg | Java API |
|---|---|---|---|
| Mix manual/generated | YES | NO | Can be implemented |
| Word® support | YES | YES | YES |
| Respect to legacy Word templates | YES | YES | YES |
| Ease of use | YES | YES | NO |
| Requires Word® installed | NO | YES | NO |
| Integration with the Java world | YES | NO, or limited | YES |
| Ability to consume any source of data | YES | NO, or limited to specific/proprietary query systems. | YES |
| Reuse, capitalization of documentation scripts | YES | NO | YES |
| Algorithmy | YES | NO, or limited | YES |
| Generators independent of Word® templates | YES | NO | NO |
| Complex tables (vertical cell merges...) | YES | NO | YES |
| Debugging | YES | NO | YES |
| Complex / heavy documents | YES | NO | NO |

## 2. Main concepts

pxDoc is a templating language dedicated to the generation of documents. There are two kinds of pxDoc components:

- **pxDocGenerator**s: they can be launched to generate documents
- and **pxDocModule**s: they can be imported by pxDocGenerators or other pxDocModules, like any java class.

Each .pxdoc file defines either a **pxDocModule** or a **pxDocGenerator**.

```
package reference.guide

pxDocModule MyPxDocModule {

  var myVar

  template myTemplate(Object model) {

  }

  function String myFunction() {
    return "hello"
  }
}
```

```
package reference.guide

pxDocGenerator MyPxDocGenerator {

  var myVar

  root template myTemplate(Object model) {

  }

  function String myFunction() {
    return "hello"
  }
}
```

As you can see, **pxDocModule**s and **pxDocGenerator**s have a lot in common. They both deal with templates, functions and variables:

- **templates**: kind of java methods dedicated to building document elements
- **functions**: they map to regular java methods
- **variables**: they map to class fields when defined in pxDocGenerators and pxDocModules, and to regular local variables when defined in templates and functions.

The only differences between pxDocGenerators and pxDocModules are as follows:

- pxDocModules cannot be executed to generate documents, but pxDocGenerators are dedicated to this task.
- pxDocModules should not import pxDocGenerators
- pxDocGenerators must define a 'root' template, which specifies the entry point for the generator.

Since a pxDoc file can define variable and functions, a pxDoc file is very close to a regular java class with the extra-ability to provide a new kind of method: the **template**s.

Inside the templates, you will be able to use the 30 document-oriented keywords that make pxDoc so fun and easy.

```
template myTemplate(Object model) {

  // start a document
  document {

    // a table of content
    toc

    // apply style Heading1
    #Heading1 { "My first title with pxDoc" }

    "Hello " color "128,128,128" { "World" } !
  }
}
```

## 3. Reusable pxDocGenerators

pxDoc provides the unique ability to design **reusable document generators** that can be applied with any Word stylesheet at generation time.


© EXTREME MODELING SOFTWARE 2018

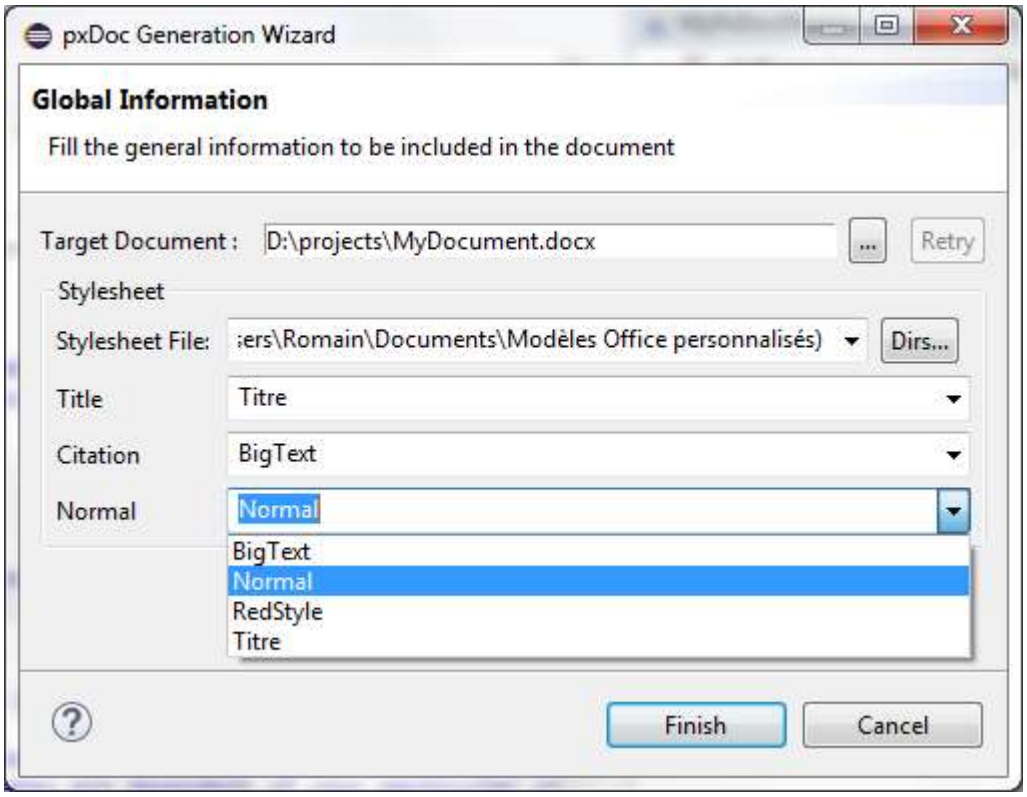
This is possible by defining 'variable' *styles* in your pxDoc modules and/or generators. Define the styles you need by just giving them a name! The following snippet defines variable styles such as *Title*, *Citation* and *Normal*:

```
pxDocGenerator MySampleGenerator
 styles: Normal, Title, Citation {
   ...
 }
```

They immediately become available for applying styles:

```
// '#' followed by a style name applies
// the chosen style to some content:
#Title {'My Title'}
#Citation {'pxDoc is easy !'}
```

At generation time, the pxDoc generation wizard will allow users to select a Word stylesheet and to bind each variable style to a style existing in the stylesheet.




> pxDoc detects only styles not tagged as *hidden* by Word. Sometimes, Word automatically tag some styles as *hidden* because they are not used.

> To ensure a style is not hidden, you just need to write some dummy text using this style in the *dotx* file. This dummy text will not appear in the generated document

## 4. Stylesheet-Dependent pxDocGenerators

If your Word stylesheet is well known and you don't want your users to select another stylesheet, (or if you simply don't need generator reusability!), you can also use your preferred and standardized Word stylesheet directly in your pxDoc templates:

```
pxDocGenerator MySampleGenerator
  stylesheet: MyDedicatedStylesheet {

}
```

You can only select a '*deployed*' stylesheet. A stylesheet can be deployed with an extension point, so that it can be used by pxDoc generators/modules.

> When starting a new 'pxDoc Project', the creation wizard proposes options to deploy new stylesheets for you.

Stylesheet-Dependent generators have both their drawbacks and advantages:

- **drawback:** your documentation templates are dependent on your particular stylesheet and the styles it defines.
- **advantage:** you don't need to care about stylesheet binding at generation time, and with automatic completion and model validation, you are sure to use the right styles!

> If you don't need to make your documentation generators reusable by others on different stylesheets, making your templates dependent on your stylesheet is not a problem!
>
> You can even change your mind later and adapt your templates so that they support variable styles. It's simple with the help of pxDoc language tooling (content-assist, validation, quick-fixes, etc.)!



© EXTREME MODELING SOFTWARE 2018

# 5. pxDoc Tooling Features

## 1. Integrate your pxDoc projects with pxGen

Projects UI generation, feature/repository generation (for update-site creation), integration capabilities...

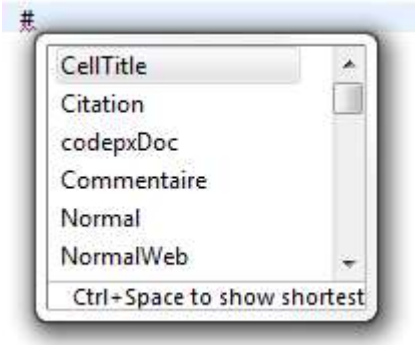
Refer to dedicated section.

## 2. Content assist

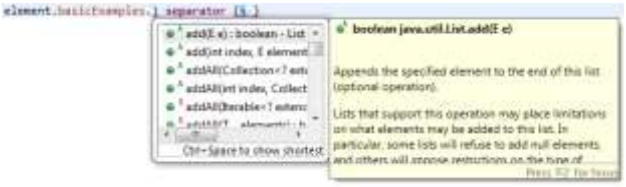
pxDoc provides content assist proposal everywhere possible.

**Colors**

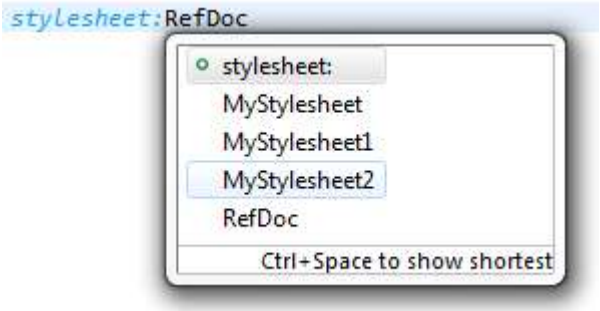
**Styles**




**Java Content-Assist**



**Deployed stylesheets**



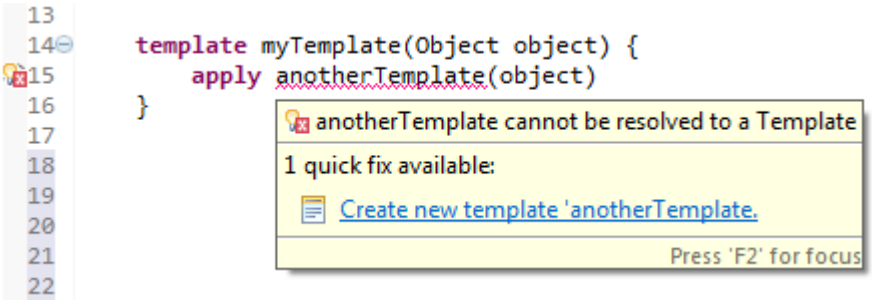

**Enum values**

## 3. pxDoc Validation & Quickfixes

pxDoc checks the validity of your document generator, and eventually provides also quickfixes to solve them.



© EXTREME MODELING SOFTWARE 2018




### 4. Java Integration

Ctrl-Shift-O

### 5. Automatic Formatting

Ctrl-Shift-F

### 6. Include external (documentation or not) dialects

External Languages integration + extensibility

## 6. Mix Manual and Generated document parts

Generating 100% of a document is not always possible, nor is it desirable.

You should eventually be able to:

- manage complex document headers (recipients, version, history of changes, signatures...), but all of this is rarely available in a model...
- complement/justify data with supplementary manual information that was not available in your data source

With pxDoc, you can insert new or pre-existing documents into your generated document.

```
// add a pre-existing document
        var String pathToSubDocument = 'path/to/subDoc.docx'
        subDocument pathToSubDocument

        // include if already exists, or initialize a new doc from a template
        var String pathToNewDocument = 'path/to/newSubDoc'  + i++ + '.docx'
        subDocument pathToNewDocument fromTemplate:'templates/template_1.docx'
```

Actually, these 'included' documents become slaves in a master/slave document relationship. They are not physically included in the generated document and remain in their own file. They can be managed under any SCM system like any other file.

pxDoc provides Word macros to expand the included documents or save standalone documents (breaking the links to the included documents and copying their content in one transportable file.)



© EXTREME MODELING SOFTWARE 2018

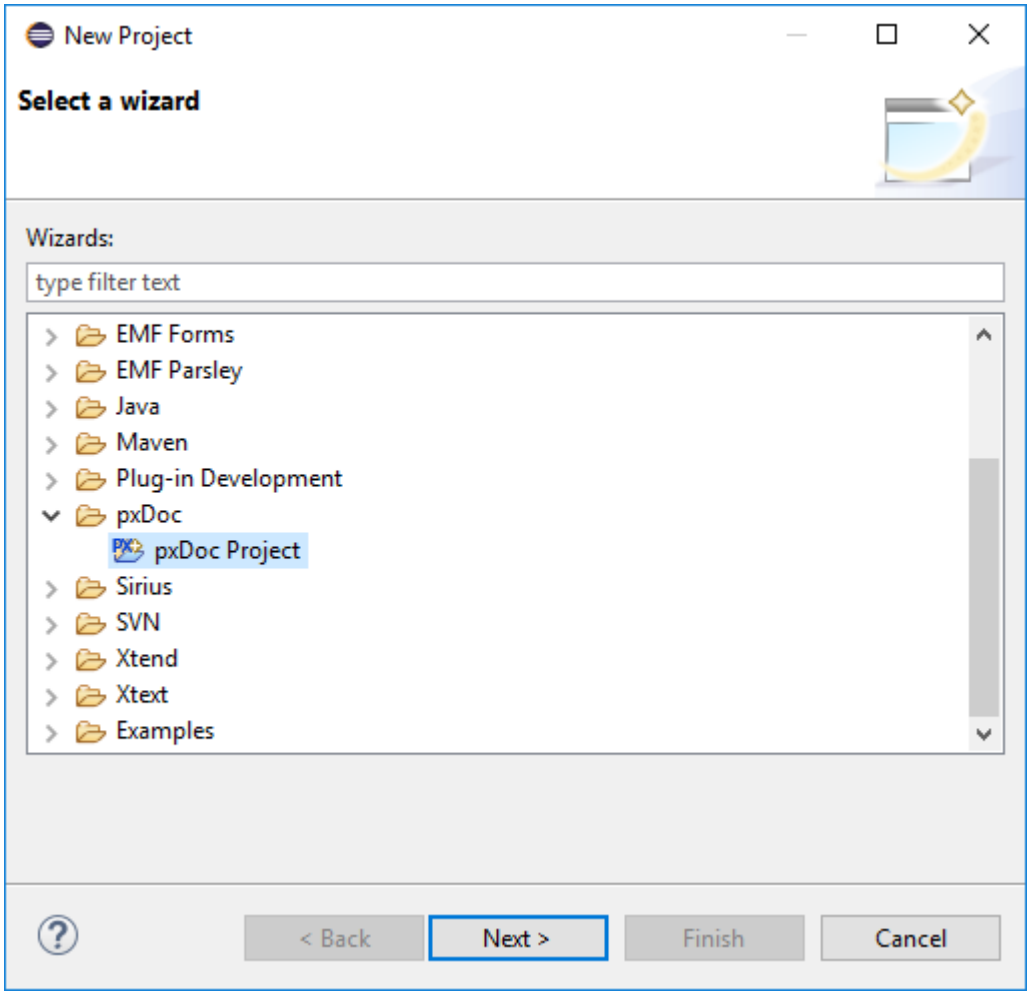
# 2. Create, publish and launch projects

pxDoc comes with several features to help the development and integration of your generators:

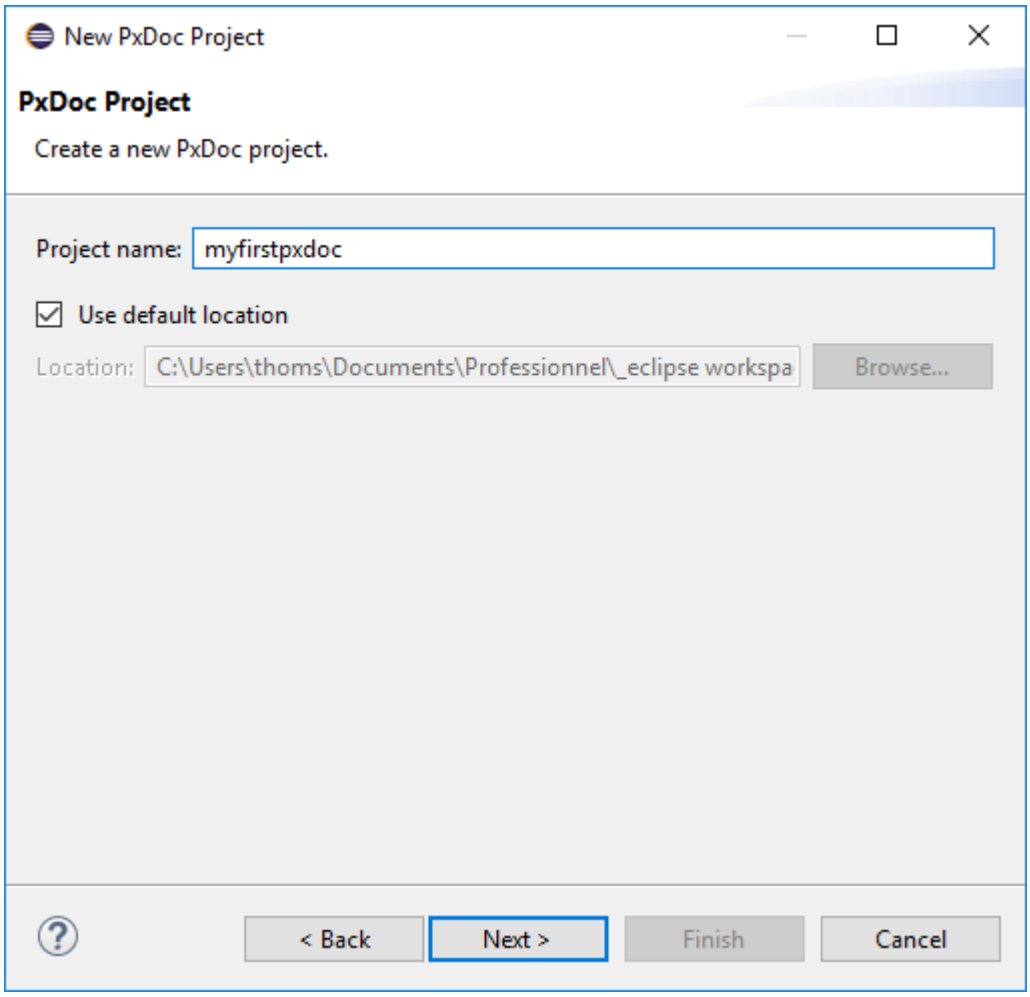
- A wizard for project creation
- A wizard to launch a document generation from an object
- pxGen tool to re-generate pxDoc runtime UI (change menu label and directory mode) or to add the generator Feature and Repository (update-site creation...)

## 1. Project creation

Create a new pxDoc project by choosing *File* → *New* → *Project…* from the main menu.




Set a valid Eclipse project name for your pxDoc project.

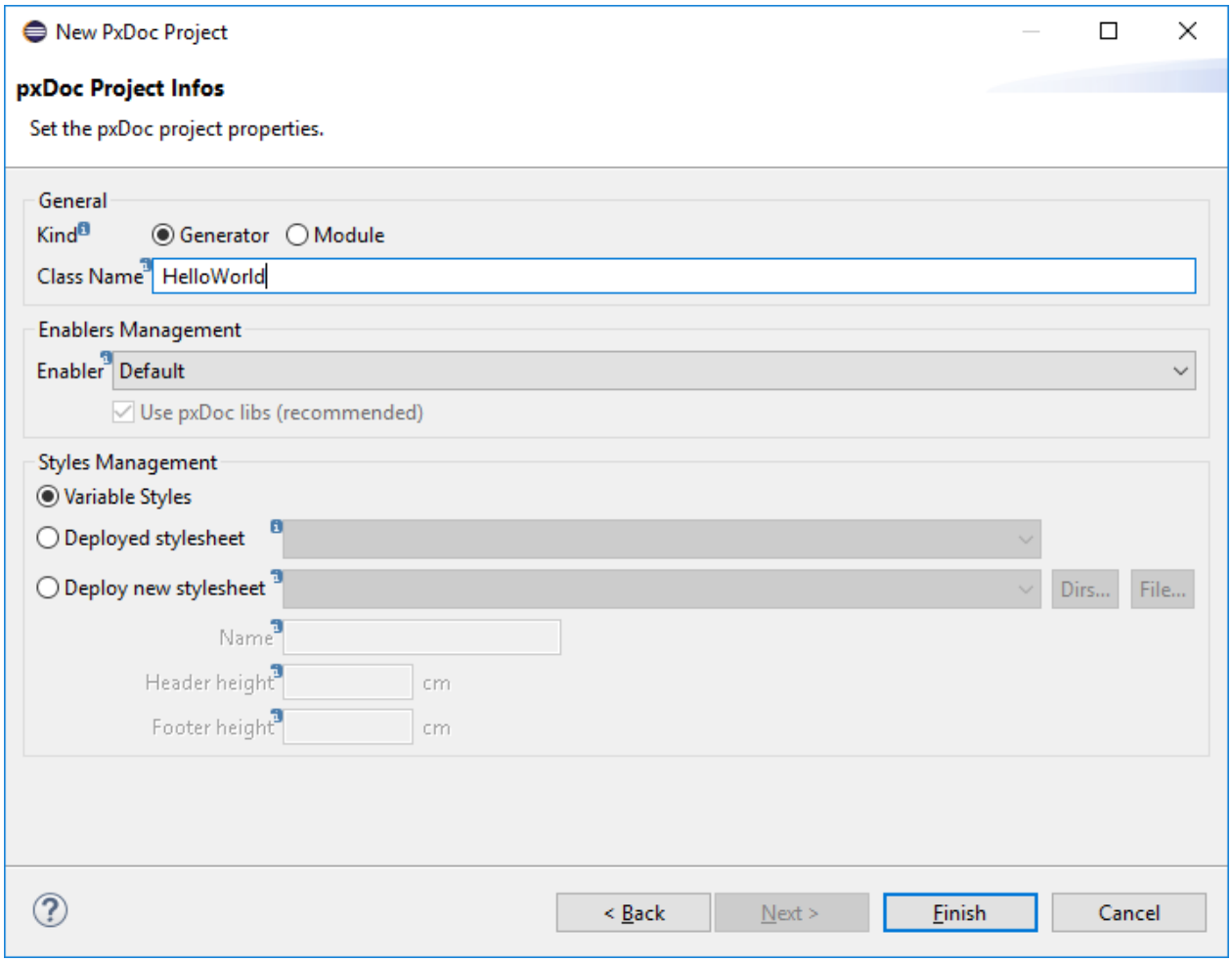



Choose *Next* to get into your new pxDoc project details:

Set a valid java class name for your generator (here "HelloWorld").



© EXTREME MODELING SOFTWARE 2018




In the **Enablers management** section, you may specify an enabler, e.g. a connector dedicated to the environment from where the generation is launched (UML tool, Modeling environment...).

> **Refer to our [Community site]** to see the enablers available and get more information about them.

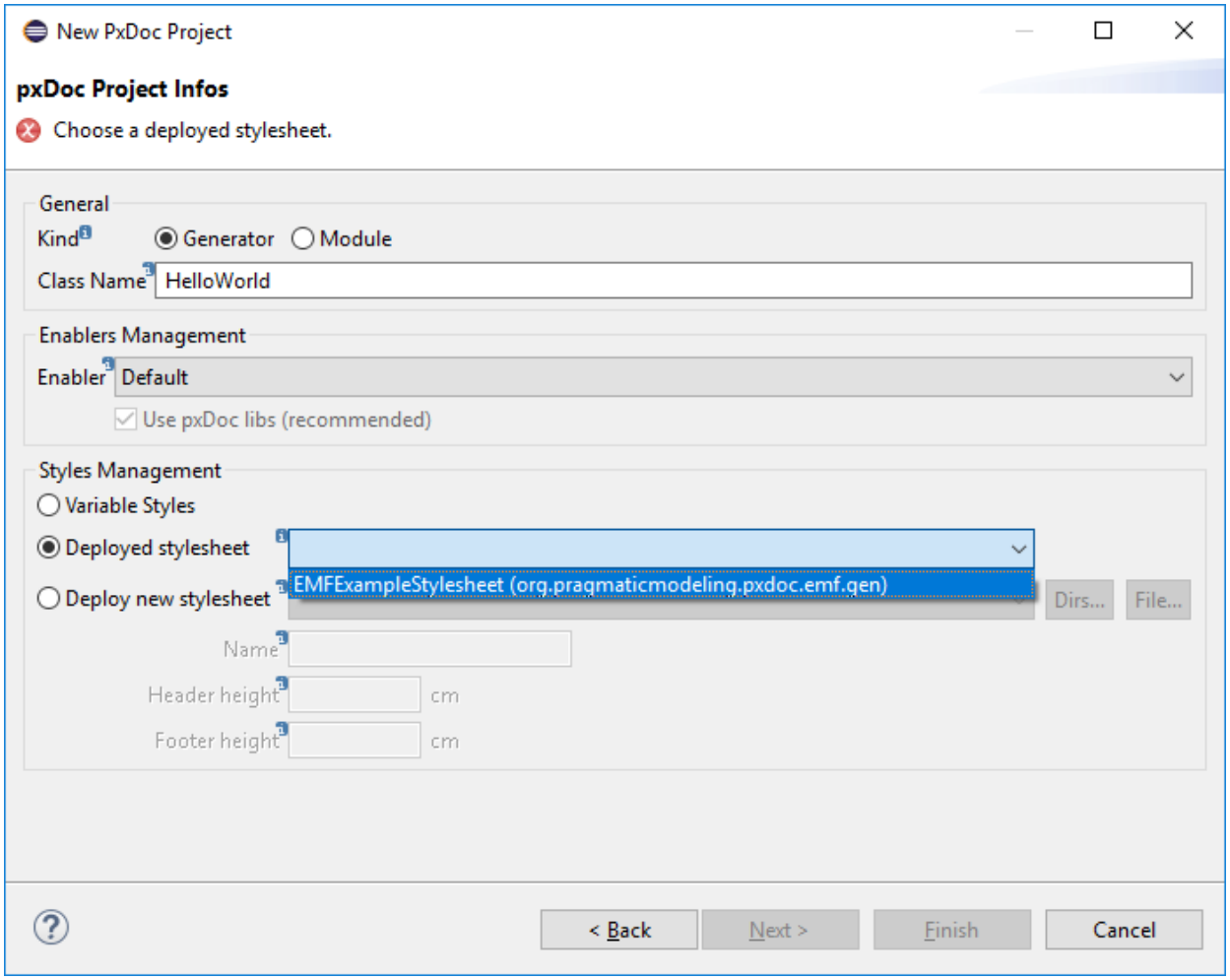
In the **Styles Management** section, select how you want to use styles (refer to the dedicated section to learn more about styles):

- Use variable styles
- Use a stylesheet already deployed in your workspace
- Use a new stylesheet, that will be available for other projects in your workspace

> If you did not setup your pxDoc preferences yet, the selection combo box will remain empty. The **Dirs...** button opens the Stylesheets directories Preferences page. Once your stylesheet directories are set, you will be able to select dotx files in the combo box.

The two parameters *Header height* et *Footer height* are used by pxDoc to calculate automatically the maximum size of images. As pxDoc does not use Microsoft Word at all, it cannot know the size of the header and footer while generating the document.

As simple example, we will use the default Enabler and the *EMFExampleStylesheet* stylesheet (deployed in the EMF generic generator that is always available in pxDoc).

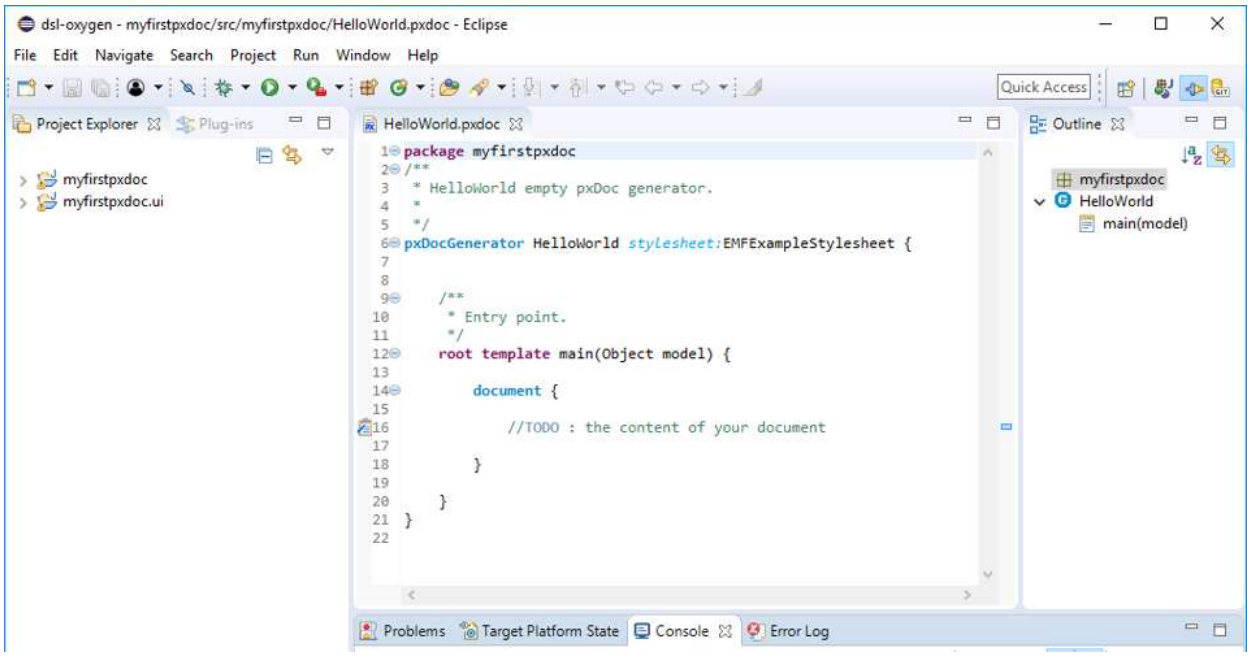



Select *Finish*.

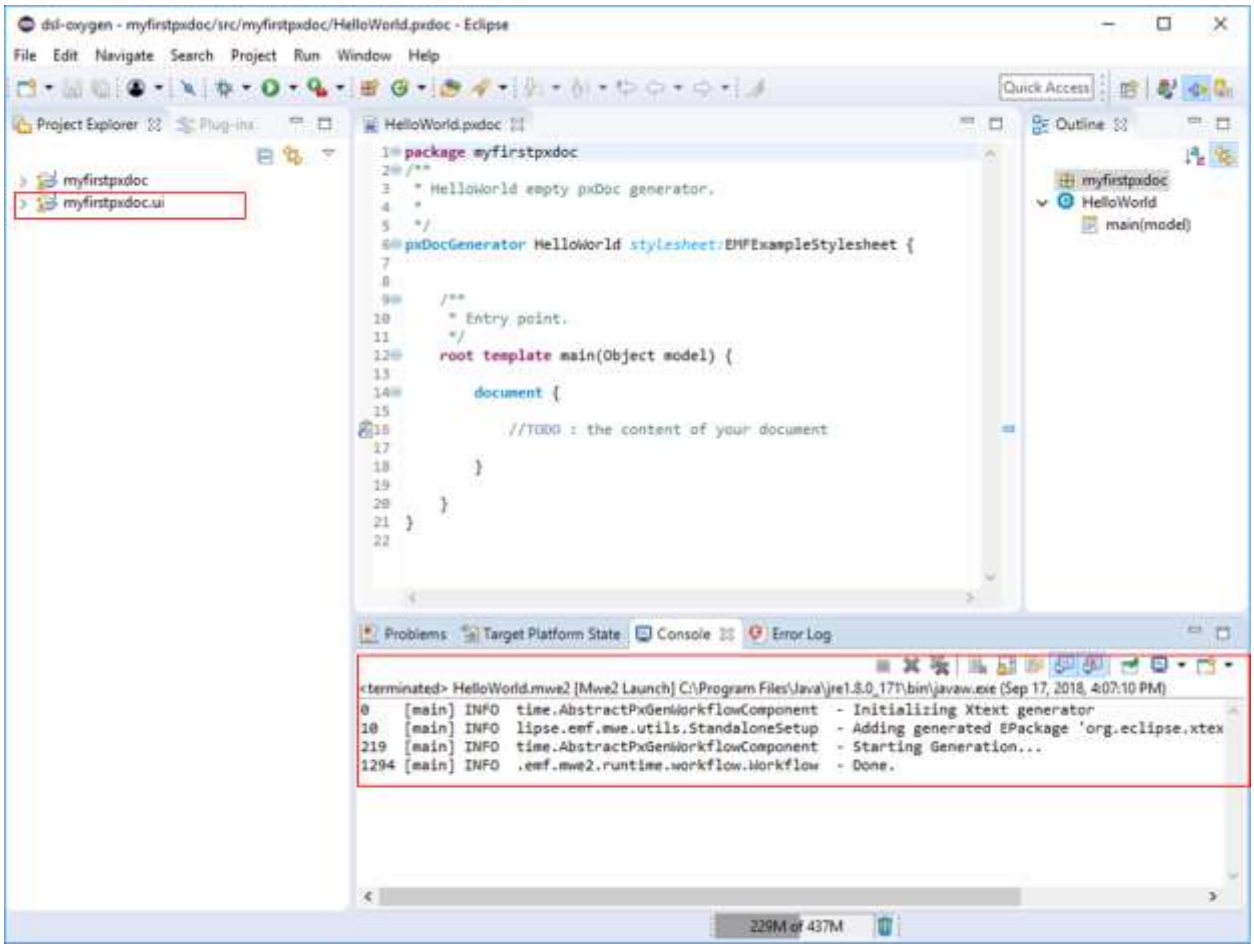
Your pxDoc project will first open with an empty pxDoc generator (.pxdoc file), and should look like the following screenshot:



© EXTREME MODELING SOFTWARE 2018




And after a little while, the UI project is generated (by the pxGen tool)



# 2. Launching the generation

To test or launch your generator:

1. select *Debug As* or *Run As -> Eclipse Application*.

> In Debug mode, the modifications made to your generator are automatically taken into account, so you just need to re-launch the generation in the 2nd Eclipse to see the result.

- Import your source model, data... in this 2nd Eclipse
- To enable the pxDoc menu, right click on a relevant object of this source (according to the command object defined in the **root template** of your Generator).

The launching wizard will pop up, with two possible configurations:

- If a stylesheet is specified in the Generator, you only have to specify the target filename (or directory)
- If variable styles are used, you also need to select a stylesheet to bind the styles of your Generator

Generator with a stylesheet defined

Generator with variable styles

© EXTREME MODELING SOFTWARE 2018

> Refer to 5min tutorial and to the 15 min tutorial to get a better view on generation launching!

## 3. pxGen

pxDoc comes with the pxGen feature which aims to:

- Generate an UI project which controls the pxDoc menu and launching wizard and includes the required enabler
- Generate Feature and/or Repository projects which will allow to publish your generator (optional).

pxGen is driven by the *.pxgen* file, located just below the Generator(*.pxdoc* file). It is **automatically launched** when a project is created to:

- create the UI project
- import some dependencies and implement some concepts (Diagram, Description Provider) when an enabler is selected and its associated common libraries are included. In the project created in the previous section, only *enable EclipseUI* is specified.

```
1  package myfirstgenerator
2
3  import fr.pragmaticmodeling.pxdoc.generator.PxDocEnablers
4  import org.pragmaticmodeling.pxdoc.enablers.eclipse.EclipseUi
5
6  generatorConf PxDocEnablers {
7
8      basePackage = "myfirstgenerator"
9      generatorClassName = "HelloWorld"
10     pxDocUri = "src/myfirstgenerator/HelloWorld.pxdoc"
11
12     /**
13  * UI eclipse project with a wizard to launch your generator.
14  */
15     enable EclipseUi {
16
17         // FIXME set your own value.
18         menuLabel = "HelloWorld"
19
20     }
21
22 }
23
```

## 1. Fine-tune the Eclipse UI

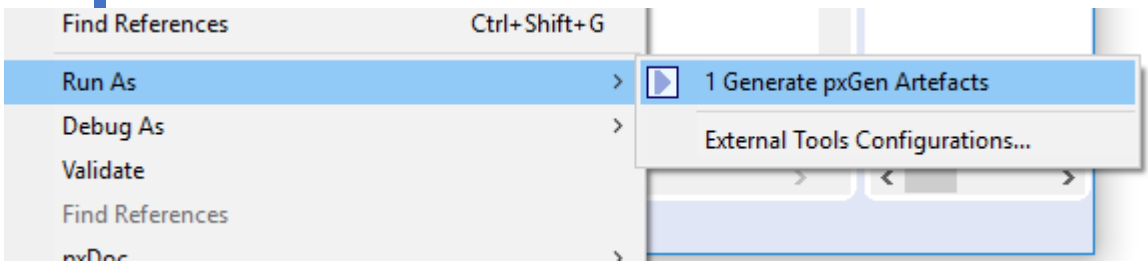
You can modify the menu item label (that will appear in the pxDoc menu)

You can also specify that you want to select a Directory and not a File as target for the generation: this is the case when a Generator produces several **document**s. To do so, add the following instruction: *selectionMode = "directory"* in the EclipseUI block.

Here is an example of an **EclipseUI** with an EMF enabler and *Directory* mode selected:

```
16    enable EclipseUi {
17
18        // FIXME set your own value.
19        menuLabel = "EMF Sample Generator"
20
21        selectionMode="directory"
22
23        enable EmfInput {
24
25        }
26    }
27
28 }
```

> You need to right-click in the file and select *Run As -> Generate pxGen Artefacts* each time you modify the pxGen file




## 2. Generate Eclipse Feature and Repository to publish the Generator

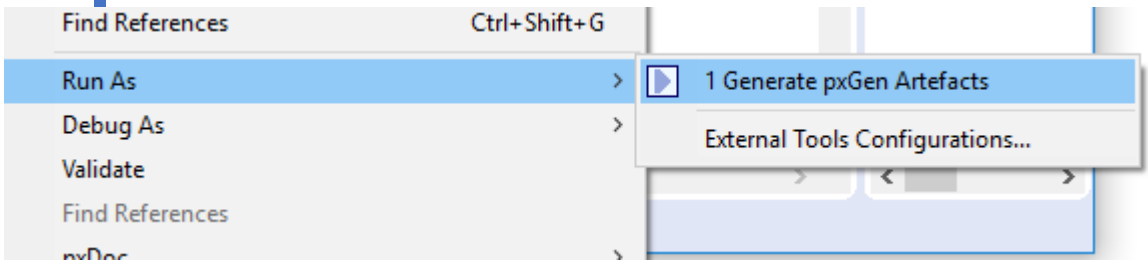
To publish your Generator as a runtime (plugin, update site...) for your end user: add the **enable EclipseFeature** and **enable EclipseUpdateSite**. If you specify them, press [CTRL]-[SHIFT]-O to add the


© EXTREME MODELING SOFTWARE 2018

required **import** to the pxGen file (you may need to do this twice).

```
1  package myfirstgenerator
2
3  import fr.pragmaticmodeling.pxdoc.generator.PxDocEnablers
4  import org.pragmaticmodeling.pxdoc.enablers.eclipse.EclipseFeature
5  import org.pragmaticmodeling.pxdoc.enablers.eclipse.EclipseUi
6  import org.pragmaticmodeling.pxdoc.enablers.eclipse.EclipseUpdateSite
7  import org.pragmaticmodeling.pxdoc.enablers.eclipse.emf.EmfInput
8
9  generatorConf PxDocEnablers {
10     basePackage = "myfirstgenerator"
11     generatorClassName = "HelloWorld"
12     pxDocUri = "src/myfirstgenerator/HelloWorld.pxdoc"
13
14     /**
15   * UI eclipse project with a wizard to launch your generator.
16   */
17     enable EclipseUi {
18         // FIXME set your own value.
19         menuLabel = "HelloWorld"
20         enable EmfInput {}
21     }
22
23     enable EclipseFeature {}
24     enable EclipseUpdateSite {}
25 }
```

> Remember to right-click in the file and select *Run As -> Generate pxGen Artefacts* each time you modify the pxGen file



# 3. Template concepts

Let's start with the few core concepts that organize your documentation generators:

- pxDocGenerator and pxDocModules,
- templates,
- functions,
- variables.

## 1. pxDocGenerators and pxDocModules

pxDocGenerator and pxDocModule share the same set of attributes.

```
pxDocGenerator <name> <attributes> { ... }
pxDocModule <name> <attributes> { ... }
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *with:*<br>module1, module2, ... | Comma-separated list of Modules that will be used by this generator | Any accessible pxDocModule name | Yes |
| *stylesheet:*<br>MyStylesheet | Name of a deployed Word stylesheet to use in the generator | Any valid deployed stylesheet ID | Yes |
| *styles:*<br>MyStyle1, MyStyle2, ... | Comma-separated list of variable style names | Only valid java identifiers (no spaces, no special characters...) | Yes |
| *styleBindings:*<br>ParamStyle1->RealStyle1, ParamStyle2->RealStyle2, ... | Comma-separated list of bindings of variable styles to concrete styles | Source styles are the Module(s) variable styles. Target styles must exist in the chosen stylesheet. All source styles must be bound. | Yes |

### Basic generator using a Stylesheet

Basic declaration of a generator which uses a stylesheet as a source for the Styles, header/footer, etc.

```
pxDocGenerator HomePageExample stylesheet:MyStylesheet {

  root template main(Object model) {
```



© EXTREME MODELING SOFTWARE 2018

```
  ...
 }
}
```

## Basic generator using variable styles

Basic declaration of a generator which uses 3 variable styles: Title1, Title2 and bulletList. These styles will have to be bound with Styles from a Word stylesheet when generating the document.

```
pxDocGenerator HomePageExample style: Title1, Titl2, BulletList {

 root template main(Object model) {
  ...
 }
}
```

## Generator with imported modules

A generator with a Module imported. The styles of the Modules are bound with the styles of the Generator stylesheet

```
pxDocGenerator HomePageExample
 with: TestModule
 stylesheet:MyStylesheet
 styleBindings:moduleTitle1->Title1, moduleTitle2->Title2 {

 /* content */
}
```

> See “Apply styles” section to learn more about style-binding.

## Declaration of a Module with a stylesheet

```
pxDocModule MyModule stylesheet:MyStylesheet {

}
```

> If you use a stylesheet in a Module, you will have to use the same stylesheet in all the Generators where this Module is imported.

### Declaration of a Module with variable styles

```
pxDocModule MyModule styles: Title1, Titl2, BulletList {

}
```

## 2. Keyword 'template'

Templates is where the magic of pxDoc happens.

Templates can be considered as regular java methods since:

- you can design your template signature (template parameters)
- you can declare local variables
- you can use almost all conditional (if-then-else), switch, or iteration instructions (for, while...).

... but with an extra set of built-in keywords dedicated to document generation:

- about resources: document, subDocument, image
- about styling: #, HeadingN, font, bold, italic, color, underline...
- about navigation: bookmark, hyperlink
- about tables: table, row, cell
- about breaks: newPage, section, nextColumn...
- and misc: table of content, field, language, samePage, ...

```
<attributes> template templateName(args) { ... }
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| **root** | Set the template to launch in a pxDocGenerator | One and only one template must be set as root in a pxDocGenerator | Yes |
| <**public**\|**private**> | Template visibility. | public, private<br>Public by default. | Yes |
| **override** | Imported templates can be overridden by a local version of a template | | Yes |



© EXTREME MODELING SOFTWARE 2018

| | | | |
|---|---|---|---|
| | with the same java erasure. | | |
| templateName | A name for the template | Any valid java method name | No |
| (Type param1, ...) | Parameters for the template | Any valid java parameter declaration expression | Yes |

## 3. Keyword 'apply'

Templates can be invoked using the special keyword **apply**.

```
apply templateName(arguments) separator {...}
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| templateName | The template to call | Any visible template. | No |
| (Type param1, ...) | Arguments given to the template | Given parameters must match the signature of a visible template | No |
| **separator** {...} | Inserts document elements (text, breaks, ...) between each template invocation | Can be used in case of template invocation with an Iterable as first parameter. Content to be included between each template invocation is set inside '{' '}' and can contain any valid document elements. | No |

## 4. Usages of template and apply keywords

### Simplest template definition

```
template myTemplate() {
  // content
}
```

## Private template

```
private template myTemplate(Person person) {
  // content
}
```

## Root template

```
root template main(Model model) {
  // content
}
```

## Simple apply of a template

Templates can be invoked using the special keyword **apply**.

```
template prettyAddress(Address address) {
  address.number " " address.street ", " address.postalCode " " address.city
}

template myTemplate(Person person) {
  var Address mainAddress = person.mainAddress
  apply prettyAddress(mainAddress) // <- call to template 'prettyAddress(Address)'
```

When the first parameter given to the template being called is an Iterable, **apply** iterates the template invocation on each element.

```
template firstTemplate(Person person) {
  // this...
  apply prettyAddress(person.addresses)

  // is equivalent to
  for (address : person.addresses) {
    apply prettyAddress(address)
  }
}
```

## Insert a separator when iterating on templates

When iterating on templates, it is sometimes useful to separate template invocations with a **separator**:

```
template firstTemplate(Person person) {
```



© EXTREME MODELING SOFTWARE 2018

```
 // the separator instruction includes a paragraph break (§) between each address
 apply prettyAddress(person.addresses) separator { § }

 // same behaviour with too much code ;-/
 for (var i=0; i<person.addresses.size; i++) {
  apply prettyAddress(person.addresses.get(i))
  if (i <person.addresses.size -1) {
   §
  }
 }

}
```

## A real-life example of template (used to generate this documentation!)

All document keywords used in this example will be explained further. Don't panic!

```
template toKeyword(Keyword keyword, int level) {

 // Specify a title with the keyword name
 HeadingN level:level {"Keyword '" keyword.keywordName "'" }

 // Insert the keyword declaration description
 apply toBasicExample(keyword.basicExamples) separator {§ }

 // Insert the introduction text for the keyword
 if (keyword.documentation !== null) {
  apply toDoc(keyword.documentation, level)
 }

 // Build the table with the keyword attributes
 if (!keyword.attributes.empty) {
  table {
   row header shadingColor:"0,0,0" {
    cell {"Attributes" }
    cell {"Description" }
    cell {"Possible values (if any)" }
    cell {"Optional" }
   }
          // Insert one row per attribute of the keyword
   apply attributeRow(keyword.attributes)
  }
  §
```

```
  }

  // Insert the conclusion text of the keyword
  if (keyword.conclusion !== null) {
    apply toDoc(keyword.conclusion, level)
  }

  // Insert the usages (examples) for the keyword
  apply toUsageExample(keyword.usageExamples)
}
```

## 5. Keyword 'function'

> All functions behaviors discussed in this section also apply to templates.

pxDoc functions are very similar to java methods:

```
function Object myFunction(String param1, boolean param2) {
  return param1 + param2;
}
```

```
<attributes> function ReturnType functionName(arguments) {...}
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| <**public**\|**private**> | Function visibility. | public, private<br>Public by default. | Yes |
| **override** | Imported functions can be overridden by a local version of a function with the same java erasure. | | Yes |
| functionName | Function name | Any valid java method name | No |
| (Type param1, ...) | Parameters for the function | Any valid java parameter declaration expression | Yes |

As pxDoc is based on Xbase, it shares with it many valuable improvements and possible code simplifications (in comparison with java).

Excerpt from the Xbase documentation:



© EXTREME MODELING SOFTWARE 2018

*Conceptually and syntactically, Xbase is very close to Java statements and expressions, with a few exceptions:*

- *No checked exceptions*
- *Everything is an expression, there are no statements*
- *Lambda expressions*
- *Type inference*
- *Properties*
- *Simple operator overloading*
- *Powerful switch expressions*

## 6. Implicit 'it' parameter

A function (or template) parameter named '**it**' will allow you some extra simplifications in your code. Remember the template prettyAddress(Address address) we took as example ?

```
template prettyAddress(Address address) {
  address.number " " address.street ", " address.postalCode " " address.city
}
```

With the **it** parameter, this template could have been written like this:

```
template prettyAddress(Address it) {
  number " " street ", " postalCode " " city
}
```

## 7. Xbaselib extensions

The org.eclipse.xtext.xbase.lib provides a set of useful extensions that make complex things easy and short:

```
function void testExtensions(List<Book> books) {
  // Few IterableExtensions
  // in our (dummy) example, Book has subclasses Novel, Fantaisy, ...

  // filter(Class) filters elements instance of the specified class
  var novels = books.filter(Novel)
```

```
 // filter can take a lambda expression [] to carry out complex (or simple!) filter conditions:
 // books published in year 1984
 var byYearBooks = books.filter[book | book.publicationYear === 1984]
 // IterableExtensions provides somes additional handy capabilities to express Integer ranges:
 var twoThousandBooks = books.filter[book | (2000..2009).contains(book.publicationYear)]

 // you can sort your Lists with sortBy... Just give the name of the attribute to make the sort
 var booksByYear = books.sortBy[publicationYear]
 var booksByTitle = books.sortBy[title]
 // you even sort using more powerful expressions
 var booksByAuthorName = books.sortBy[author.name]
}
```

> We recommend you to take time and learn more about extensions provided by *org.eclipse.xtext.xbase.lib*. They will allow you to make great things with your input data.

## 8. var / val

Variables can be declared with the var and val keywords.

- **var** is used to declare non-final variables.
- **val** is used to declare final variables. This is useful when you need to access a variable from a closure expression.

> Variables have a public visibility by default. You can set them private using the private keyword.

Variables can be defined:

- directly under **pxDocGenerator** or **pxDocModule**. In this case, they map to public fields under the underlying generator/module generated java classes.
- inside a pxDoc **template** or **function**. In this case, they map to a local variable of the underlying generated java method.

```
<var|val> <type>? name (= <initial>)?
```



© EXTREME MODELING SOFTWARE 2018

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| <**var**\|**val**> | Declare a variable | **var**: non-final variables<br>**val**: final variables | No |
| <type> | The (java) type of the variable | Any valid java type.<br>Can be omitted since Xbase language support type inference | Yes |
| name | Name of the variable | Any valid java variable name. | No |
| = <initial> | An initial value for the variable | Any valid xbase/java expression. | Yes |

## In templates and functions:

```
var Type name = ...
val Type name = ...
```

## In pxDocGenerators and pxDocModules, you can also set the variable visibility:

```
private var String privateString = "Hello"
public val String publicFinalString = "World"
```

# 4. Document concepts

## 1. Documents and Subdocuments

### 1. Keyword 'document'

A document means a Word “docx” file that will be generated by pxDoc. You can declare one or several document(s) in any template. But the most common case is that you will have only one document in the **root template** of your Generator.

> If several documents are defined in a generator, do not forget to add the *selectionMode='Directory'* instruction to your pxGen file. This will allow you to select a directory, and not a file, as target location for the generation

**document** <attributes> { ... }

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *fileName:* "filename.docx" | Specify the name for the generated document.<br><br>This attribute is mandatory only when several documents are declared in a single generator | Any valid file name. Path not allowed here: it will be specified in the generation launching wizard. | No |
| *properties:* prop1="propValue", prop2=true, ... | A comma separated list of properties affectations. | Any property defined in the stylesheet (if the generator is dedicated to a specific stylesheet), or any new property name and values that will be stored as custom properties in the generated document. | Yes |
| *attachedTemplate:* "path/to/file" | Path to a Word template, that may include macros, …<br><br>Path must be accessible by all user opening the | Any docx, docm, dotm file. | Yes |



© EXTREME MODELING SOFTWARE 2018

| | | | |
|---|---|---|---|
| | document, the macros are not embedded in it. | | |

## Set variables and properties or the document

You can set several variables and properties for a document, each one being separated with a comma:

```
document properties:creator="Thom Johnson", revision=1 variables:variableName="My variable value",
var2="\\To06\Projects\ModelDocs\"
{
  // document content
}
```

> Document properties (Word standard and custom from your stylesheet) can be suggested with [CTRL]+[SPACE].

## Define a filename and a template for the document

When several documents are generated, you need to specify a filename for each document. You can also specify the path to an existing document containing some macros to be used in the generated document. Below, we define two documents and one of them has a template with macros:

```
pxDocGenerator MyGenerator stylesheet:MyStyleSheet {
  root template main(Object model) {

    document fileName:"c:\myDoc1.docx" {
      "This is a simple document"
    }

    document
      attachedTemplate: "templateWithMacros.dotm"
      fileName:"c:\myDocWithMacros.docx" {

      "This is another document, which will provide some macros from the template"
    }
  }
}
```

> The macros are not embedded in the document: pxDoc generates *docx* but not *docm* files. The *dotm* file specified as template must be accessible by the user opening the document, to get the macros available.

## 2. About Sub-Documents and manual parts

You can specify slave Word documents (also called sub-documents) into the master document being generated.

With pxDoc, you can use it in two different ways:

- Basic reference to an existing external document
- Creation of a "manual part" in your document (more details below)

**Focus on manual parts:**

Manual parts are sections that will not be erased when you re-generate the document. They are managed through subdocuments.

The first time the document is generated, subdocuments are created (in the same directory or wherever you want). At the next generations, pxDoc will detect that these subdocuments already exist and will not change them.

## 3. Keyword 'subDocument'

**subDocument** <attributes>

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| **subDocument** "path/to/subdoc.docx" | The first parameter is the relative or absolute path of the file to include. | Any valid file name expressed in relative or absolute. | No |
| *fromTemplate:* "templates/file.docx" | Path to the Word document used to initialize the manual part, when they do not already exist. | Path to an existing Word document. | Yes |

### Link to an existing subdocument

With an absolute path:



© EXTREME MODELING SOFTWARE 2018

```
subDocument "\\MyComputer\MyDocuments\mySubDoc.docx"
```

Or relative to the generated document:

```
subDocument "MyDocuments/mySubDoc.docx"
```

### Define manual parts in an iterative way

You can create manual parts iteratively: here, in the context of an impact analysis on some applications, we want to insert comments and results for each application based on the template "templateImpactAnalysis.doc".

The subdocuments are created in the subdirectory "ImpactAnalysis" of the generated document locationand have the application name in their filename.

The following code will be inserted in a loop (for each application...):

```
var String filename = "ImpactAnalysis\Impact Analysis of " + appli.name + ".docx"
subDocument filename fromTemplate: "templates/templateImpactAnalysis.docx"
```

# 2. Import content from other languages

## 1. Keyword 'language'

The language keyword is used to automatically transform a content from a given structured external language (HTML, wiki...) into the pxDoc language.

> Today, two renderers are provided with pxDoc: HTML and pxDoc source code. With time, we will integrate more converters.
>
> You can extend an existing language renderer. You can also develop your own renderer. Refer to the dedicated section *pxDoc Extensibility/Develop your own language renderer* to get more details.
>
> Do not hesitate to share you language renderers or extensions on our Community site!

```
language [<lang-id>|class:<lang-id>] extension:ext "expression-to-parse"
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|

| | | | |
|---|---|---|---|
| <lang-id> | Name of a language supported by an installed language renderer. | Any language name supported by a language renderer. Today, 'html' and 'pxdoc' are supported out-of-the-box. Not used if class: is specified. | No |
| *class:*<lang-id> | Name of a language supported by an installed language renderer and provided as a String by a variable, function... | Anything providing a language name as a String (variable, function...). Not used if <lang-id> is specified (hard-coded). | No |
| <expression_to_parse> | A string expressed in a specific language. | | No |
| extension:<ext> | Take into account an extension of for the specified language renderer | Any java instance implementing *IPxDocRendererExtension* as an extension for the language renderer | Yes |

### Include html content in several ways

```
var anHtmlExpression = myApi.getHtmlDocumentation()
language html anHtmlExpression
var languageType="html"
language class:languageType anHtmlExpression
language html
 "<ul>
  <li>item1</li>
  <li>item2</li>
 </ul>"
```

### Use an extension of the HTML renderer

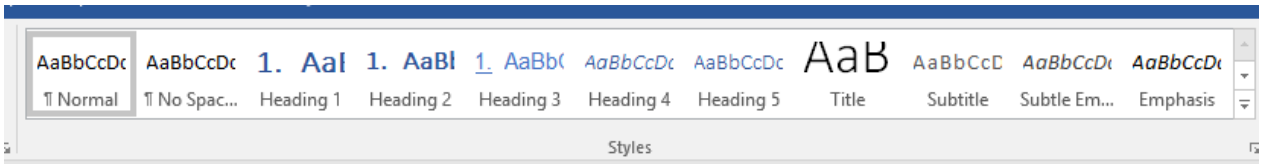
Refer to the documentation describing the generic Generators

## 3. Applying styles

What we call styles are the Styles you can define in Word:



© EXTREME MODELING SOFTWARE 2018

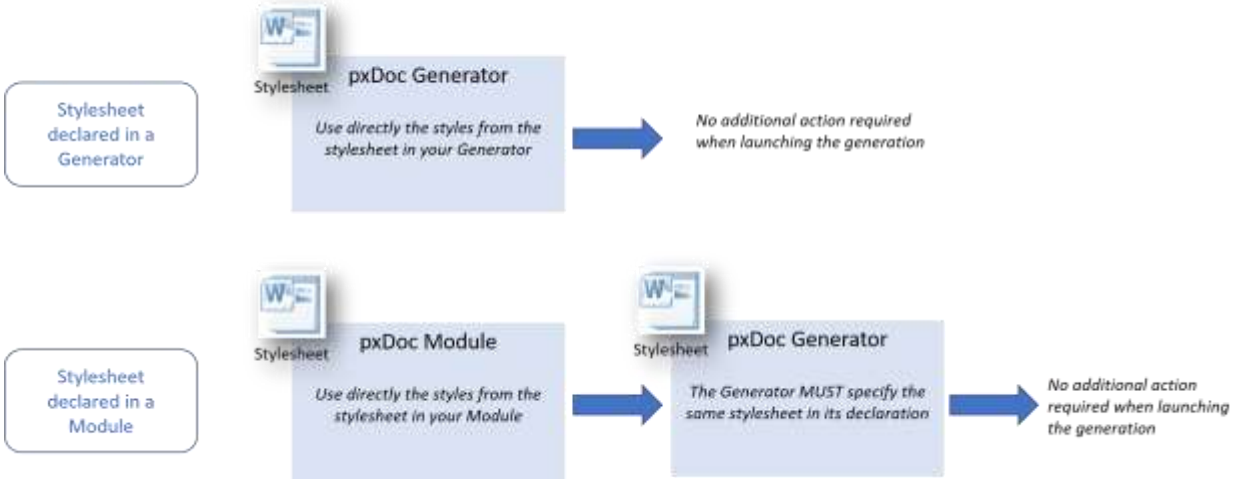



You can directly apply these styles to the content of your document (see Keyword #, below).

There are 2 ways to declare styles in a Generator or a Module: get styles directly from a stylesheet, or define “variable styles”.

## 1. Get Styles from a stylesheet

This is the easiest and most efficient way for simple generators: you get the styles from a Word Stylesheet (.dotx) that is referenced in your pxDoc project when creating it.




The stylesheet is specified when creating the project and it can:

- be located in an existing project in your workspace
- copied from an external location into your project: they are then referenced as an extension in the plugin.xml file of your project.

```
pxDocGenerator MyGenerator stylesheet:MyStylesheet {...}
```

Same with a Module:

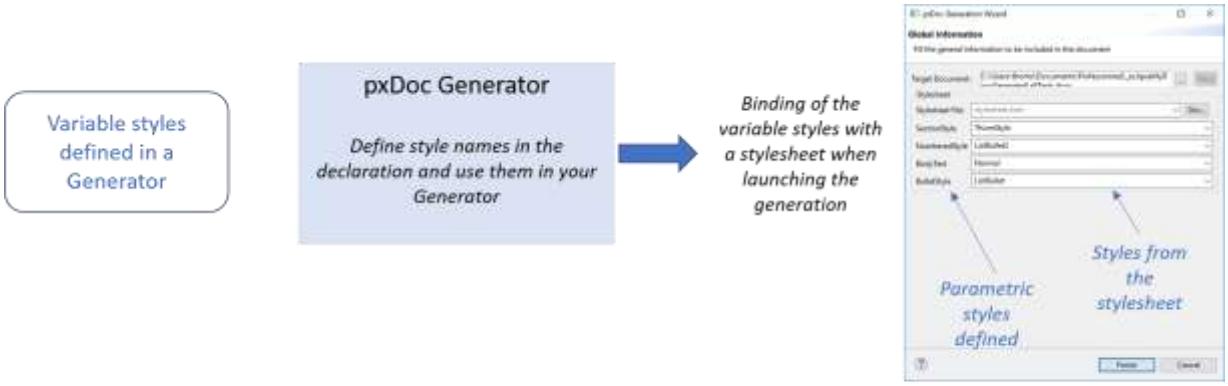
```
pxDocModule MyModule stylesheet:MyStylesheet {...}
```

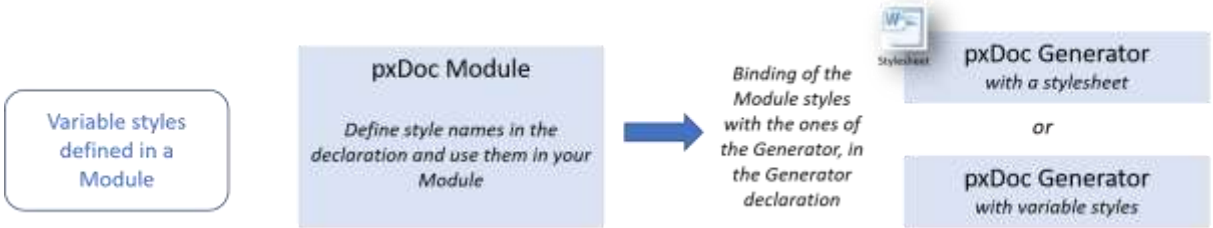
> If you use a stylesheet in a Module, you will need to use the same stylesheet in your Generator. So, the most generic/reusable way is to define variable styles in your Modules

## 2. Use variable styles

Variable style are style-names defined directly in the generator. These styles do not have any formatting characteristics: they will be bound with a stylesheet in the end.

- Variable styles from a Generator will be bound manually with the styles of a stylesheet when launching the document generation
- Variable styles from a Module will be bound in the Generators:
    - Either with the Styles from the stylesheet declared in the Generator
    - Or with the variable styles declared in the Generator




To define variable styles, use the attributes styles: when declaring a Module or a Generator to declare the style-names.

### Use variable styles in a Module

A Module is created with the styles myTitle1, myTitle2:

```
pxDocModule TestModule styles: myTitle1, myTitle2
```

When declaring a Generator using this module, you map these styles with the ones of the Generator (here, from a stylesheet):


© EXTREME MODELING SOFTWARE 2018

```
pxDocGenerator TestGenerator
  with: TestModule
  stylesheet:"stylesheet.dotx"
  styleBindings: myTitle1->Heading1, myTitle2->Heading2 {
  ...
}
```

### 3. Keyword 'styleApplication'

To apply a style to a content, just use the hash '#' before the style-name:

```
#Heading1 {"My Title 1"}
#Normal {"My content in Normal"}
```

> You can use the suggestions [CTRL]+[SPACE] from Eclipse to see the available styles from the stylesheet or from the list of variable styles.

> pxDoc detects only styles not tagged as *hidden* by MS Word. Sometimes, Word automatically marks some styles as *hidden* because they are not used.
>
> To ensure a style is not hidden, you just need to write some dummy text using this style in the stylesheet. This dummy text will not appear in the generated document

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *align:* <left\|...> | Set the paragraph alignment | both, centered, left, right | Yes |
| *keepWithNext* | Keep this paragraph with the next (avoid page break). | | Yes |
| *numbering* (*level:*<int>) | Apply a numeric list to the style | Level: integer between 1 and 6 | Yes |
| *bullet* (*level:*<int>) | Apply a bullet list to the style | | Yes |
| *shift:* <Measure> | Set an indentation for the paragraph | refer to About Measures section | Yes |

### Apply a Style

You can directly write text:

```
#Heading2 {"The Style Heading2" }
```

Or get content from a function:

```
#Normal {"This the class: " class.getName() }
```

### Specify attributes to the style

```
#Normal align:center {"You can specify the alignment" }
#Heading2 shift:2cm {"You can also specify the indent of the text"}
```

> To learn more about the unities you can use: refer to About Measures section.

## 4. Keyword 'HeadingN'

You can use the HeadingN keyword to apply heading styles (from level 1 to 9), which are particular Styles in Word.

```
HeadingN level:<int>
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *level:* <int> | Set the title level | Integer between 1 and 9 | No |
| *keepWithNext* | Keep this paragraph with the next (avoid page break). | | Yes |

### Use HeadingN in a recursive algorithm to get the accurate title level

HeadingN is particularly useful when creating different levels of title in your algorithms.

For example, below, we want to generate a description of nested packages:

```
template pkgDocumentation(Package pkg, int titleLevel) {
 HeadingN level:titleLevel {pkg.name}

 ... /* documentation of this package */

 for (subPkg : pkg.nestedPackages) {
```



© EXTREME MODELING SOFTWARE 2018

```
    // We apply recursively the current template to the nested packages,
    // increasing the Heading level (e.g. subtitles) with each level of nested packages
    apply pkgDocumentation(subPkg, titleLevel+1)
  }
}
```

We have defined a template for a package description, which contain:

- A package and a titleLevel as argument
- A title of level 'titleLevel' showing the package name
- A description of this package
- A loop to call to the same template for the nested packages for which the title will be at level titleLevel+1

In the generator, we just need to apply the template to the top-level package:

```
var Package myPackage = ...
apply pkgDocumentation(myPackage, 1)
```

# 4. Modifying styles

Obvious keywords have been defined to apply basic style modifiers to your text:

- Unitary modifiers: keywords **bold**, **italic**, **underline**, **color**,...
- Keyword font to directly apply several modifiers

These styling are applied to a block and can be nested.

## 1. Keyword 'bold'

```
bold {...}
```

Apply **bold** style to the bloc

## 2. Keyword 'italic'

```
italic {...}
```

Apply *italic* style to the bloc

### 3. Keyword 'underline'

Apply underline style to the bloc

**underline** *type:*<single|dash|...> {'underlined'}

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *type:*<single\|dash\|...> | The type of underline | single, dash, dashDotDotHeavy, dashDotHeavy, dashedHeavy, dashLong, dashLongHeavy, dotDash, dotDotDash, dotted, dottedHeavy, double, thick, wave, wavyDouble, wavyHeavy, word | Yes |

### 4. Keyword 'strike'

Apply strike style to the bloc

**strike** *type:*<single|DOUBLE> {'striked text'}

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *type:*<single\|DOUBLE> | The type of strike | single, DOUBLE | Yes |

### 5. Keyword 'subscript'

**subscript** {...}

Apply $_{\text{subscript}}$ style to the bloc

### 6. Keyword 'superscript'

**superscript** {...}

Apply $^{\text{superscript}}$ style to the bloc

### 7. Keyword 'color'

Apply a color to the bloc



© EXTREME MODELING SOFTWARE 2018

> Use [CTRL]-[SPACE] to pick up a color directly in a pop-up window!

**color** "R,G,B" {...}

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| "R,G,B" | The RGB value as a comma separated string | R, G and B are integers between 0 and 255. | No |

## 8. Keyword 'font'

You can use the keyword font to set the font name and size, the background color and strike/underline type

**font** <attributes> {...}

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *name:* fontName | Name of the font to be applied | Any font name that is available in the stylesheet<br>With variable styles, no restriction (no stylesheet to parse available fonts) | Yes |
| *size:* fontSize<br>*size:* 12 | Size of the font in points | integer | Yes |
| **bold** | Set the font to bold | | Yes |
| **italic** | Set the font to italic | | Yes |
| **subscript** | Set the font to subscript | | Yes |
| **superscript** | Set the font to superscript | | Yes |
| **underline** (<br>*type:* <single\|...> ) | Set the font underlined | | Yes |
| **strike** (<br>*type:*<single\|...> ) | Set the font striked | | Yes |
| *color:* "R,G,B" | Set the font color | R, G and B are integers between 0 and 255. | Yes |
| *backgroundColor:* "R,G,B" | Set the font background color | | Yes |

# 9. Usages

## Use Styles and basic style modifiers

```
#Title {"My first document with some basic styles" }
 #Normal {
  "You can set"
  color "255,0,0" {" the color of the text." }
  italic {" It is quite easy to write in italic" }
  bold {" or in bold" }
  underline {" or underlined" }
  " or"
  bold {
   italic {
    underline {
     " with all these 3 styles."
    }
   }
  }
 }
 #Normal {
  font name: "Times New Roman" size:12 color: "0,0,255"
   backgroundColor: "192,192,192" bold italic underline strike(SINGLE)
  {
   "Who wants to apply all these styles at once?!?"
  }
 }
```

With our Stylesheet from the tutorial, you will get this:

© EXTREME MODELING SOFTWARE 2018

# 5. Pages and line breaks, Sections

## 1. Keyword 'paragraphBreak'

**§**

Insert a paragraph break.

## 2. Keyword 'newLine'

**newLine**

Insert a line break.

## 3. Keyword 'nextColumn'

**nextColumn**

Insert a column break, e.g. go to the next column in a section with multiple columns. (see **section** below)

## 4. Keyword 'newPage'

**newPage**

Insert a page break

## 5. Keyword 'section'

Define a new section in the document.

**section** <attributes>

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *type:* <continuous\|...> | Define the type of the section | continuous<br>evenPage<br>oddPage<br>nextPage | Yes |
| *landscape* | Change to landscape orientation (incl. page break, of course) | When omitted, a section is considered as Portrait per default | Yes |
| *column:* <int> | Define the number of columns in the section | Any integer | Yes |

| | | | |
|---|---|---|---|
| *columnsSpace:* <measure> | Define the empty space between 2 columns | refer to About Measures section | Yes |

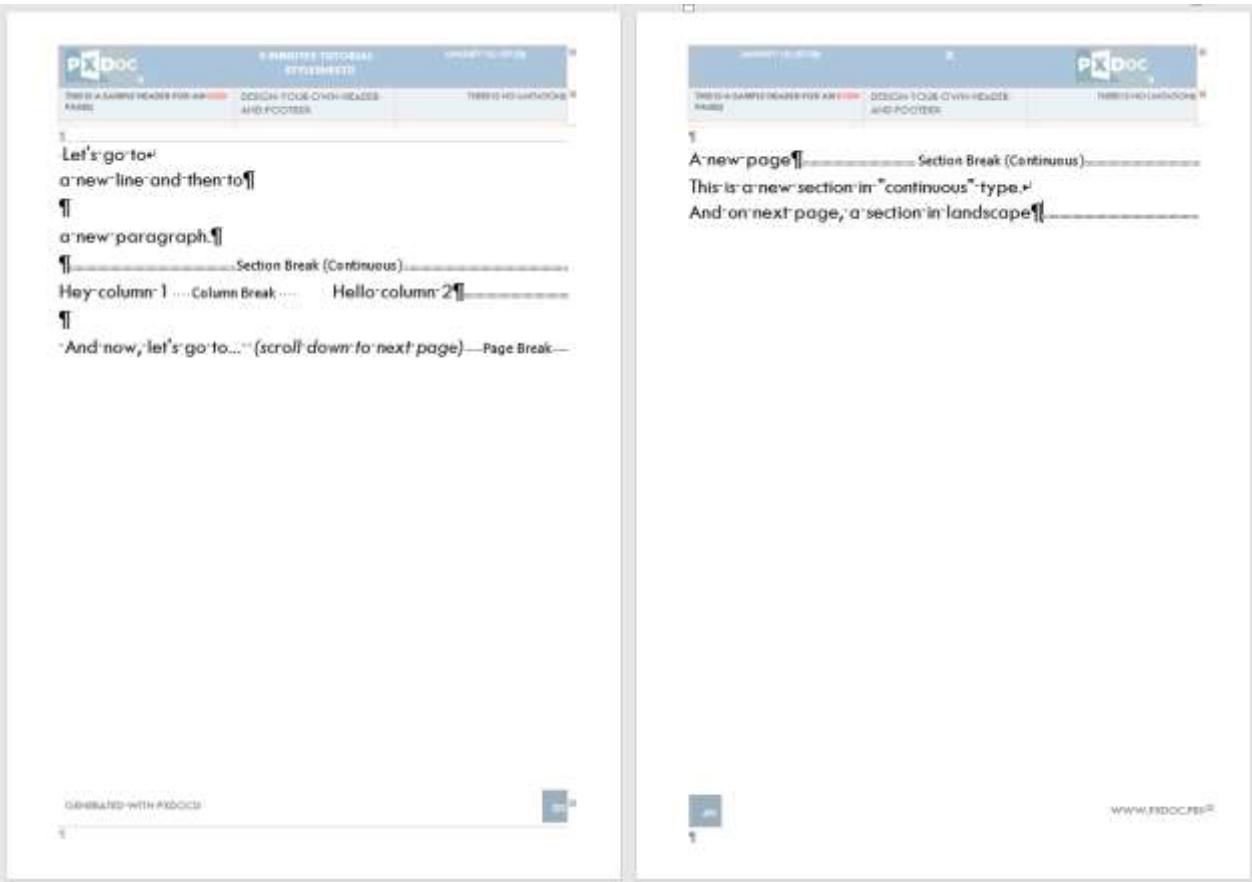
## 6. Breaks usages

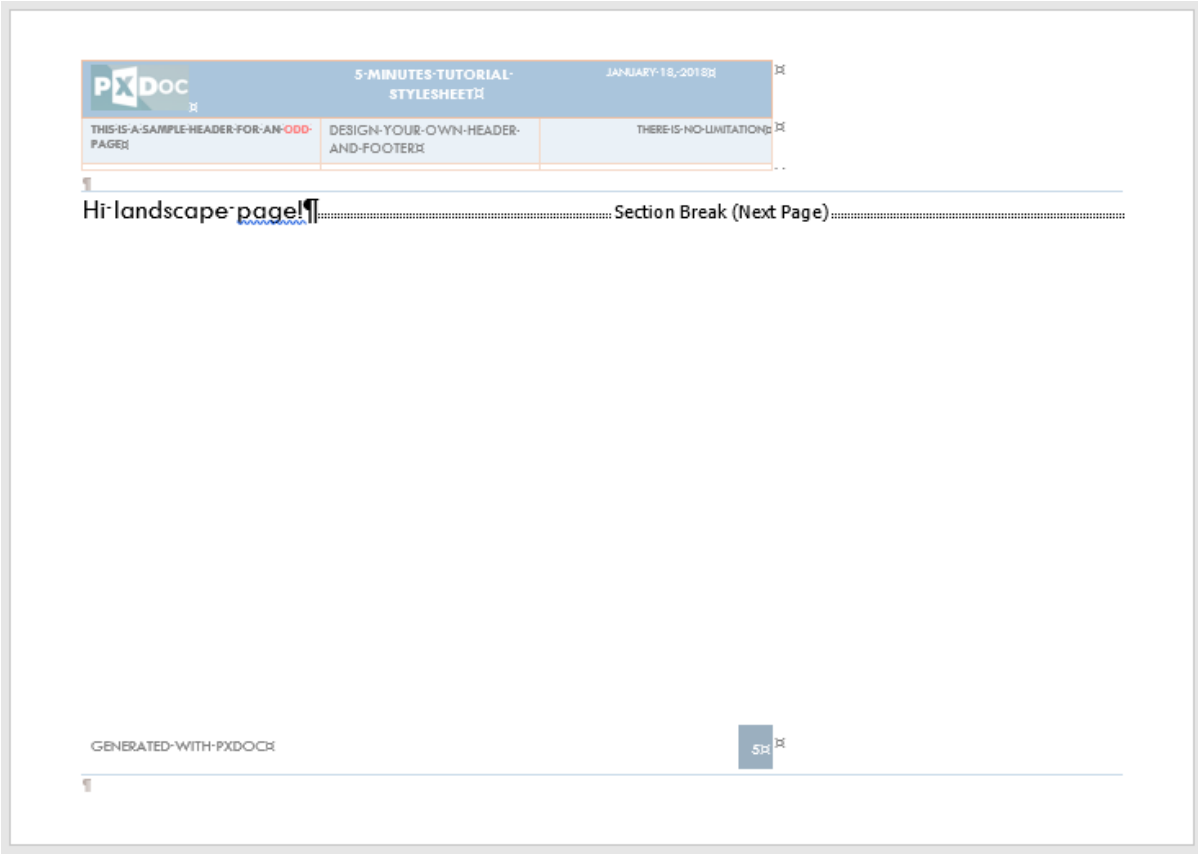
### Try all section types and breaks

```
"Let's go to"
newLine
"a new line and then to"
§
"a new paragraph."
§
§
section columns:2 type: continuous
"Hey column 1"
nextColumn
"Hello column 2"
section columns:1 type: continuous
§
§
" And now, let's go to... "newPage
"A new page"
section type:continuous // per default, this means Portrait
"This is a new section in continuous type."
newLine
"And on next page, a section in landscape"
section landscape
"Hi landscape page!"
section
```



© EXTREME MODELING SOFTWARE 2018


Let's go to
a new line and then to
a new paragraph.
Section Break (Continuous)
Hey column 1
Column Break
Hello column 2
And now, let's go to... (scroll down to next page)
Page Break
A new page
Section Break (Continuous)
This is a new section in "continuous" type.
And on next page, a section in landscape




## 7. Keyword 'samePage'

The samePage instruction aims to keep its content in the same document page. This is particularly useful when inserting a set of images that need to be kept on the same page.

> Warning
>
> SamePage is more an 'experimental' function that may not work properly 100% of time (e.g. when there is a long text to display in the block).
>
> But it is a useful function in most cases!

```
samePage {
  HeadingN level:2 {"Illustration " }
  #Normal {"The images below are too big for the page, but it has been automatically resized by pxDoc!"
}
```



© EXTREME MODELING SOFTWARE 2018

```
 image path:"images/too_big.png"
}
```

> As pxDoc generate native docx file, without 'using' Microsoft Word itself, it cannot know the height of the header and footer when generating the document: these heights depend on the content of the headers and footers defined in your stylesheet or template.
>
> To solve this concern, you need to provide those values to pxDoc, using the headerHeight and footerHeight attributes when deploying a new stylesheet. You can modify these values in the *plugin.xml* file in the project where the stylesheet was first deployed.

# 6. Include images

## 1. Keyword 'image'

Insert an image:

- Either from an existing file
- Or from a java.awt.Image object

```
image <attributes>
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *path:* imagePath | Define the location of the image. The path can be local to your project or a URL/path on your PC/network/internet. | Cannot be used with the 'java:' attribute | No |
| *java:* imageObject | Insert an image with a java object | Any java.awt.Image. Cannot be used with the 'path:' attribute | No |
| *width:* <measure> | Specify the width of the image | refer to About Measures section | Yes |
| *height:* <measure> | Specify the height of the image | | Yes |

| | | | |
|---|---|---|---|
| *hPosition* (<br>*align:* <left\|...><br>*relativeFrom:* <page\|...> ) | Set the horizontal position of the image | **align:** left, right, center<br>**relativeFrom:** character, column, page, margin, insideMargin, outsideMargin, leftMargin, rightMargin | Yes |
| *vPosition* (<br>*align:* <top\|...><br>*relativeFrom:* <page\|...> ) | Set the vertical position of the image | **align:** bottom, top, center, inside, outside<br>**relativeFrom:** line, page, margin, insideMargin, outsideMargin | Yes |

## 2. Usages of image

### Image embedded in your project

```
image path:"images/an_image.png"
```

> If an image is too big for the page (height or width), pxDoc automatically resize it.

```
image path:"images/an_image.png" width:100pixels
image path:"images/an_image.png" width:5cm
```

> You can specify only the height (or width) of the image,pxDoc automatically calculate the width (or height) to keep the aspect ratio.

### Image on your computer

An image on your computer which takes 50% of the available height.

```
image path:"C:/images/myImage.png" height:50pc
```

### Image in line with the text, at the top left of the paragraph

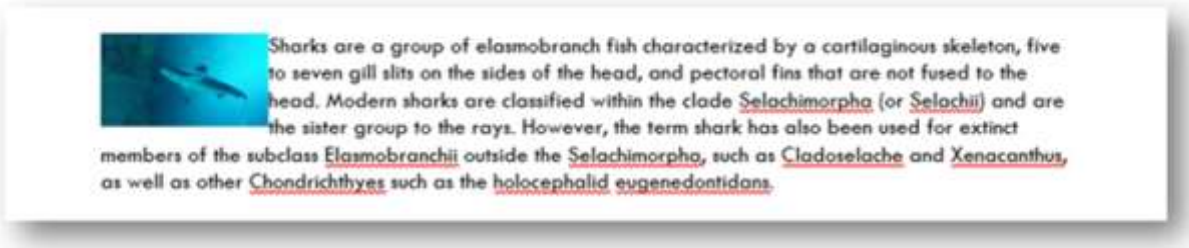
You can specify the position of the image



© EXTREME MODELING SOFTWARE 2018

```
#Normal{
  image path:"images/aquarium1.jpg" width:3cm  hPosition(align:left relativeFrom:margin)
vPosition(align:top relativeFrom:paragraph)
  "Sharks are a group of elasmobranch fish characterized by a cartilaginous skeleton, five to seven gill slits on the sides of the head, and pectoral fins that are not fused to the head. Modern sharks are classified within the clade Selachimorpha (or Selachii) and are the sister group to the rays. However, the term ""shark"" has also been used for extinct members of the subclass Elasmobranchii outside the Selachimorpha, such as Cladoselache and Xenacanthus, as well as other Chondrichthyes such as the holocephalid eugenedontidans."
}
```

You get the following result:




## Image horizontally and vertically centered

Changing the positioning to:

- Horizontal: centered relatively from the margins
- Vertical: centered relatively to the line:

```
#Normal{
  image path:"images/aquarium1.jpg" width:3cm  hPosition(align:center relativeFrom:margin)
vPosition(align:center relativeFrom:line)
  "Sharks are a group of elasmobranch fish characterized by a cartilaginous skeleton, five to seven gill slits on the sides of the head, and pectoral fins that are not fused to the head. Modern sharks are classified within the clade Selachimorpha (or Selachii) and are the sister group to the rays. However, the term ""shark"" has also been used for extinct members of the subclass Elasmobranchii outside the Selachimorpha, such as Cladoselache and Xenacanthus, as well as other Chondrichthyes such as the holocephalid eugenedontidans."
}
```

You get the following result:

Sharks are a group of elasmobranch fish characterized by a cartilaginous skeleton, five to seven gill slits on the sides of the head, and pectoral fins that are not fused to the head. Modern sharks are classified within the clade Selachimorpha (or Selachii) and are the sister group to the rays. However, the term shark has also been used for extinct members of the subclass Elasmobranchii outside the Selachimorpha, such as Cladoselache and Xenacanthus, as well as other Chondrichthyes such as the holocephalid eugenedontidans.

# 7. Include bookmarks and hyperlinks

## 1. Keyword 'bookmark'

A bookmark has a name (ID) and is applied to a block of text.

```
bookmark <id> {...}
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| <id> | The bookmark id. Once defined, hyperlinks can target bookmarks in your document. | Any bookmark value that is accepted by the Word standard: must start with an alphabetic character, should not exceed 40 characters, no special characters allowed. | No |

### Define a bookmark

```
HeadingN level:1 {bookmark "intro" {"Introduction"}}

bookmark getBookmarkId(myObject) { myObject.name }
```

## 2. Keyword 'hyperlink'

Hyperlink is used to insert a link to a bookmark or to a location outside the document. It is applied to a block of text (text to display) and may contain a comment.



© EXTREME MODELING SOFTWARE 2018

**hyperlink** <attributes> {...}

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *bookmark:* "<bookmarkID>" | Insert a link to a bookmark in the document Not used if an hyperlink is specified | Any valid bookmark ID | No |
| *href:* "URL" | Insert a link to a web page, a folder or a filet Not used if a bookmark is specified | Any valid path or URL (URL must begin with “http(s)://” | No |
| *comment:* "a comment" | Alt text, displayed when flying over the link | | Yes |

### Link to a bookmark in the document

Link to the bookmark “intro” with a display text:

**hyperlink** *bookmark:*"intro" {"Go back to the introduction"}

### Link to a web site

Link to a website with a given display text and a tooltip when flying over the link:

**hyperlink** *href:*"http://www.pxdoc.fr" *comment:*"pxDoc is really cool!" {**italic** {"A link to our website" }}

# 8. Fields and Table of Content

## 1. Keyword 'toc'

To insert a simple table of content use the keyword **toc**. You can add a title and parameters for the table of contents.

> The parameters are the ones available for the TOC field in MS Word. Refer to the OOXML specification for the complete set of available TOC parameters.

**toc** <attributes>

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *title:* "title" | A title of the TOC | Any string | Yes |
| *titleStyle:* styleName | Apply a style to the title | An existing style (variable or explicit) | Yes |
| *parameters:* "parameters-expression" | Specify advanced parameters for the TOC | See the OOXML specification for managed parameters. | Yes |

### TOC with a title and showing only the headings from level 1 to 3

Insert a TOC with the title “This is the Table of Content” (on which the style Heading1 is applied).

The parameter \o “1-3” means that the TOC will present the headings from level 1 to 3 only.

```
toc title:"This is the Table of Content" titleStyle:Heading1 parameters:'\o "1-3" '
```

## 2. Keyword 'field'

You can insert Word fields in your document, use the keyword field.

```
field <FIELDNAME> <attributes>
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| <ADDRESSBLOCK\|...> | Name of the field to insert. | ADDRESSBLOCK, ADVANCE, ASK, AUTHOR, AUTOTEXT, AUTOTEXTLIST, BIBLIOGRAPHY, CITATION, COMMENTS, COMPARE, CREATEDATE, DATABASE, DATE, DOCPROPERTY, DOCVARIABLE, EDITTIME, FILENAME, FILESIZE, FILLIN, FORMCHECKBOX, FORMDROPDOWN, FORMTEXT, GOTOBUTTON, GREETINGLINE, HYPERLINK, IF, INCLUDEPICTURE, INCLUDETEXT, INDEX, KEYWORDS, LASTSAVEDBY, LINK, LISTNUM, MACROBUTTON, MERGEFIELD, MERGEREC, MERGESEQ, NEXT, NEXTIF, | No |


© EXTREME MODELING SOFTWARE 2018

| | | | |
|---|---|---|---|
| | | NOTEREF, NUMCHARS, NUMPAGES, NUMWORDS, PAGE, PAGEREF, PRINT, PRINTDATE, PRIVATE, QUOTE, RD, REF, REVNUM, SAVEDATE, SECTION, SECTIONPAGES, SEQ, SET, SKIPIF, STYLEREF, SUBJECT, SYMBOL, TA, TC, TEMPLATE, TIME, TITLE, TOA, TOC, USERADDRESS, USERINITIALS, USERNAME, XE | |
| *parameters:* "parameters-expression" | Parameters expression suited to the selected field. | See OOXML specification. | Yes |
| *display:* "pretty-text" | Text to display until field is refreshed in Word | Any string | Yes |

### Insert the page number

```
field PAGE
```

### Insert the name of the author of the document

Insert the field AUTHOR, give the value “Thomas S” to the property of the document, display the name in upper case:

```
field AUTHOR parameters:' "Thomas S" \* Upper'
```

Another way to show the name of the author with a text to display until the field is refreshed:

```
field DOCPROPERTY parameters:"Author" display:"Refresh field to display the Author name"
```

# 9. Tables

## 1. Principles

the pxDoc language is designed to simplify the generation of tables inside documents. At its most simple expression a table can be specified with:

```
table {
 row {
  cell {"Cell1" }
  cell {"Cell2" }
  cell {"Cell3" }
 }
 row {
  cell {"Cell4" }
 }
 row {
  cell {"Cell5" }
  cell {"Cell6" }
 }
}
```

Without any explicit indications, pxdDoc will infer table dimensions: the table width is set to the page available width, and each cell share the available width with its siblings in the row.

And the result will be:

| | | |
|---|---|---|
| Cell1 | Cell2 | Cell3 |
| Cell4 | | |
| Cell5 | Cell6 | |

Properties can be injected into the table at several levels:

```
table <global-properties> [ <column-1-properties> | <column-2-properties> | ... ] {

 row <global-properties> [ <column-1-properties> | <column-2-properties> | ... ] {

  cell [ <cell-1-properties> ] { "Cell Content" }
  cell [ <cell-2-properties> ] { "Cell Content" }
 }
}
```



© EXTREME MODELING SOFTWARE 2018

```
table <global-default-properties>
    [ col1-props | col2-props | ... ]   {
        row <global-default-properties>
            [ col1-props | col2-props | ... ]   {
                cell [cell-properties]      cell [cell-properties]
            } // end of row
        row <global-default-properties>
            [ col1-props | col2-props | ... ]   {
                cell [cell-properties]      cell [cell-properties]
            } // end of row
    } // end of table
```

- global table properties will be applied by default to every cell, except if superseded at row or cell level.
- column properties expressed at table level will be applied by default to column child cells, except if superseeded at row or cell level.
- global row properties will be applied by default to every cell in the row, except if superseded at row column properties level or cell level.
- row column properties will be applied accordingly to row cells, except if superseded at cell level.
- cell properties can override any inherited properties from row or table level

## 2. Keyword 'table'

```
table <attributes> [ <col1-props> | <col2-props> | ... ] {
  //definition of rows
}
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *width:* <measure> | Width of the table. | | Yes |

| | | | |
|---|---|---|---|
| *left:* <measure> | Insert a left margin for the table. | refer to About Measures section | Yes |
| *shadingColor:* "R,G,B" | Set a shading color for the entire table. | RGB comma separated value. | Yes |
| *shadingPattern:* <solid\|...> | The shading pattern to apply. | Use suggestion [CTRL]-[SPACE] to see all available patterns | Yes |
| *borders* (<br>*size:* <int><br>*type:*<single\|...><br>*color:*"R,G,B" ) | Set border properties for all borders (bottom, top, left, right) of all the cells of table. | - size: allows to set the size of the all borders in twips<br>- type: allows to set the kind of borders<br>- color: allows to set the color of the borders | Yes |
| [ <col1-props> \| <col2-props> \| ... ] | Set table columns properties. See next section for details. | | Yes |

## 3. Keyword 'columnProperties'

Column attributes can be defined at table and row levels, with override principle (cell attributes overriding any table, row or column attributes).

They are defined between square brackets, with a vertical bar as a separator between columns:

```
[ <col1-properties> | <col2-properties> | ...]
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| *width:* <measure> | Set the column width. | refer to About Measures section | Yes |
| *vAlign:* <bottom\|...> | Set the vertical alignment of the content of cells | bottom, center, top | Yes |
| *shadingColor:* "R,G,B" | Set a shading color for the column. | RGB comma separated value. | Yes |
| *shadingPattern:*<solid\|...> | The shading pattern to apply to the column. | Use suggestion [CTRL]-[SPACE] to see all available patterns | Yes |
| *merge:*<start\|...> | Set merge behavior for column's cells. | - none: no merge, or stop merge<br>- start: starts merging cells | Yes |


© EXTREME MODELING SOFTWARE 2018

| | | | |
|---|---|---|---|
| | | - proceed: continue to merge cells<br>- automatic: this option analyzes cell contents and merge cells having the same content. This is the simplest option to merge cells, but it is sometimes necessary to control the merging process using the basic none, start and proceed values. | |
| *bottomBorder* (<br>*size:* <int><br>*type:*<single\|...><br>*color:*"R,G,B" ) | Set a bottom border for column's cells.<br>Bottom border properties can be specified to manage the border size, type and color. | - size: allows to set the size of the border in twips<br>- type: allows to set the kind of border<br>- color: allows to set the color of the border | Yes |
| *topBorder* (<br>*size:* <int><br>*type:*<single\|...><br>*color:*"R,G,B" ) | Set a top border for column's cells.<br>Top border properties can be specified to manage the border size, type and color. | | Yes |
| *leftBorder* (<br>*size:* <int><br>*type:*<single\|...><br>*color:*"R,G,B" ) | Set a left border for column's cells.<br>Left border properties can be specified to manage the border size, type and color. | | Yes |
| *rightBorder*( *size:*<int><br>*type:*<single \| dash \| ...><br>*color:*"R,G,B") | Set a right border for column's cells.<br>Right border properties can be specified to manage the border size, type and color. | | Yes |
| *borders* (<br>*size:* <int><br>*type:*<single\|...><br>*color:*"R,G,B" ) | Set all border properties (bottom, top, left, right) for column's cells. | - size: allows to set the size of the all borders in twips<br>- type: allows to set the kind of borders<br>- color: allows to set the color of the borders | Yes |

## 4. Keyword 'row'

Specify a row in a table.

```
row <attributes> [<col1>|<col2>|...] {
  //definition of cells
}
```

| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| **header** | Specify the current row is a header row.<br>It is useful for some functionalities (repeat header row at each page…) | | Yes |
| *cantSplit* | Specify that the row cannot be displayed on 2 different pages | | Yes |
| *height:* <measure> | Specify the height of the row | refer to About Measures section | Yes |
| *shadingColor:* "R,G,B" | Set a shading color for the entire row. | RGB comma separated value. | Yes |
| *shadingPattern:* <solid\|...> | The shading pattern to apply. | Use suggestion [CTRL]-[SPACE] to see all available patterns | Yes |
| *borders* (<br>*size:* <int><br>*type:*<single\|...><br>*color:*"R,G,B" ) | Set border properties (bottom, top, left, right) for row's cells. | - size: allows to set the size of the all borders in twips<br>- type: allows to set the kind of borders<br>- color: allows to set the color of the borders | Yes |
| [ <col1-props> \| <col2-props> \| ... ] | Set row's columns properties | Refer to the columnProperties section | Yes |

## 5. Keyword 'cell'

Defines a cell in a row.



© EXTREME MODELING SOFTWARE 2018

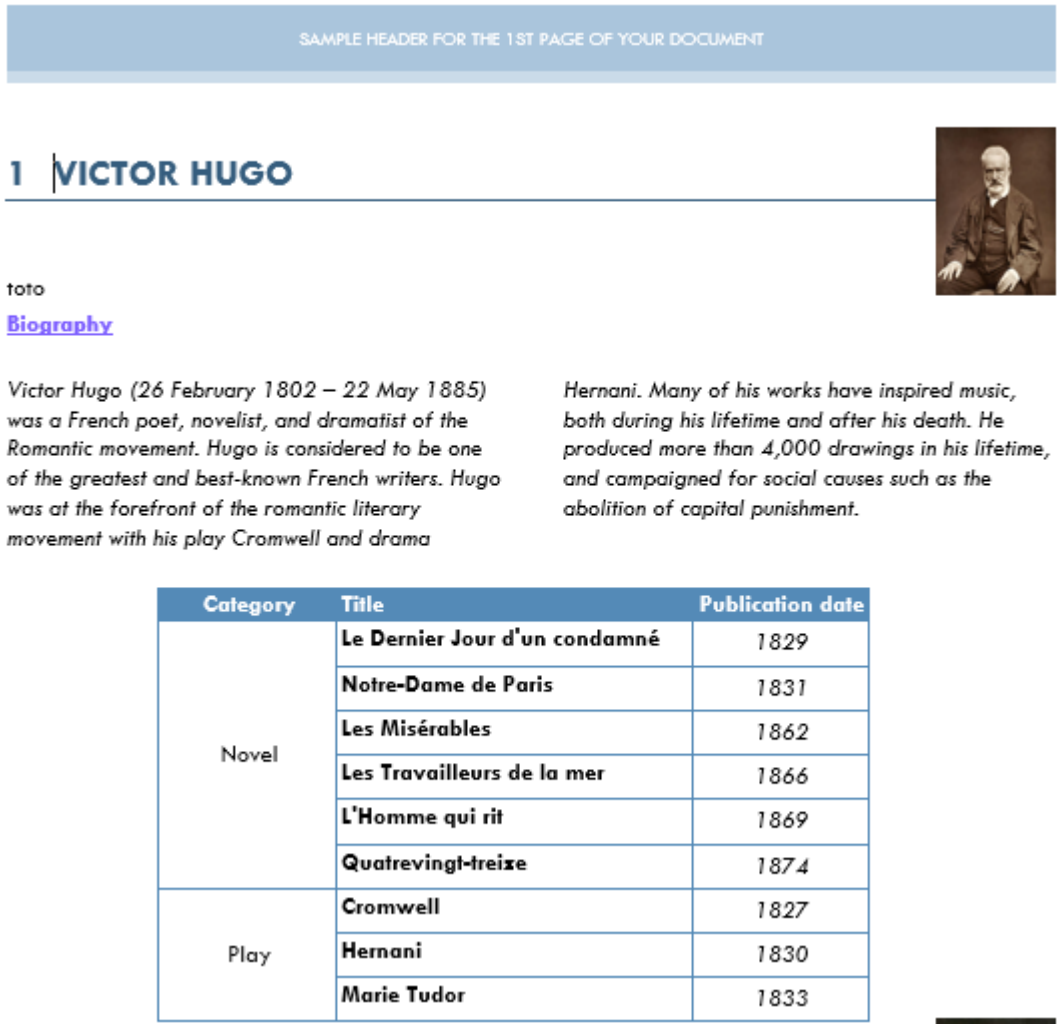
| Attributes | Description | Possible values (if any) | Optional |
|---|---|---|---|
| [ <cell-props> ] | Set the cell properties. | Same as the columnProperties . | Yes |

## 6. Usages

### A table with automatic merge

Let's look at the example shown on the homepage of our website. The purpose is to create a table with a bibliography of some authors with some cells merged in the Category column.

Example for Victor Hugo:



SAMPLE HEADER FOR THE 1ST PAGE OF YOUR DOCUMENT

1 VICTOR HUGO

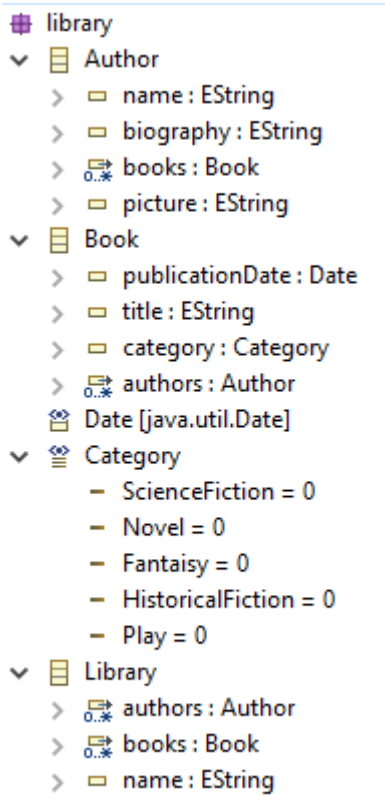
toto

Biography

*Victor Hugo (26 February 1802 – 22 May 1885) was a French poet, novelist, and dramatist of the Romantic movement. Hugo is considered to be one of the greatest and best-known French writers. Hugo was at the forefront of the romantic literary movement with his play Cromwell and drama Hernani. Many of his works have inspired music, both during his lifetime and after his death. He produced more than 4,000 drawings in his lifetime, and campaigned for social causes such as the abolition of capital punishment.*

| Category | Title | Publication date |
|---|---|---|
| Novel | Le Dernier Jour d'un condamné | 1829 |
| | Notre-Dame de Paris | 1831 |
| | Les Misérables | 1862 |
| | Les Travailleurs de la mer | 1866 |
| | L'Homme qui rit | 1869 |
| | Quatrevingt-treize | 1874 |
| Play | Cromwell | 1827 |
| | Hernani | 1830 |
| | Marie Tudor | 1833 |

This example is based on the following metamodel:




In this example, we simply used the merge:automatic attribute: pxDoc will automatically merge the cells when it detects that the books have the same categories:

```
table left:2.5cm borders(size:8 type:single color:"84,138,183"  ) [ width:20pc | width:40pc | width:20pc ]
{

 // Definition of the heading row
  row {
     cell {"Category"}
     cell {"Title"}
     cell {"Publication date"}
   }
   for (book : author.books.sortBy[category]) {
     row {
        cell [merge:automatic] {book.category}
        cell {book.title}
        cell {book.publicationDate}
     }
   }
}
```



© EXTREME MODELING SOFTWARE 2018

> Have a look at our Table Cookbook to get more examples of table design.

## 10. About Measures

You can express measures in several unities available in Word.

- Centimeters: cm
- Pixels: pixel
- Twips (twentieth of a point): twips
- Percentage: pc (100% corresponds to the whole available width or height of the page)

# 5. pxDoc extensibility

## 1. Common extensibility tasks on the UI project

In this section, we will present some of the extensibility capabilities of pxDoc, based on examples:

- Adapt the Command class of the UI project to directly bind variable styles with a stylesheet
- Adapt the generation launching wizard to your needs

### 1. Adapt the generation launching wizard

#### 1. Introduction

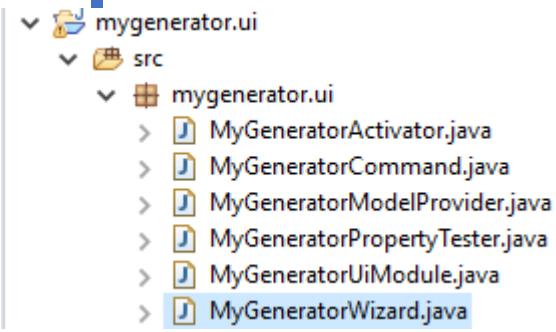
You might want to provide many generation parameters for your document generator, and users must be able to feed these generation parameters in the generation wizard (e.g. Title, Company name and Version of the document in the following examples).

You will taylor the generation launching wizard to propose this option

#### 2. Adapt the Wizard class

First, you will add a page to the wizard by adding some code to the Wizard class

> The Wizard class can be found in the *src* folder of the UI project linked to your generator.




Adapt the following code to your generator name, to add a new page to the generation wizard:

```
package mygenerator.ui;

/**
 * Wizard class
 */
```


© EXTREME MODELING SOFTWARE 2018

```
public class MyGeneratorWizard extends AbstractMyGeneratorWizard {

	@Override
	public void addPages() {
		super.addPages();
		addPage(new OptionPage("Custom Options"));
	}

}
```

3. Create a new page in the Wizard

Now, you will create the new Wizard page containing text fields for the required properties in a class named *OptionPage*.

> Create a java file in the *src* folder of the UI project, and make it inherit from *WizardPage* and implement *IPxDocWizardPage*.

You will need to provide an implementation for 3 methods of the interface *IPxDocWizardPage*:

- void fillLauncher(PxDocLauncher pLauncher): this method will be called for every IPxdocWizardPage of the generation wizard, filling your parameter values in the PxDocLauncher object.
- IDialogSettings buildDialogSettings(): builds a IDialogSettings, that can be provided as default options for the next generation.
- void loadDialogSettings(IDialogSettings pSettings): loads the previously built and saved IDialogSettings and initialize page widgets accordingly.

The code below defines the input text fields and send the content as properties to the pxDoc Launcher.

```
package mygenerator.ui;

import org.eclipse.jface.dialogs.DialogSettings;
import org.eclipse.jface.dialogs.IDialogSettings;
import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

import fr.pragmaticmodeling.pxdoc.runtime.launcher.PxDocLauncher;
import fr.pragmaticmodeling.pxdoc.runtime.ui.eclipse.IPxDocWizardPage;
import fr.pragmaticmodeling.pxdoc.runtime.ui.eclipse.wizard.IPxDocWizard;
```

```
import fr.pragmaticmodeling.pxdoc.runtime.util.PxDocLauncherHelper;

/**
 * Wizard class
 */

public class OptionPage extends WizardPage implements IPxDocWizardPage {

	private static final String SETTINGS_ID = "SampleOptionsPage_Settings";

	private static final String SOCIETY = "society";

	private static final String DOCUMENT_TITLE = "title";

	private static final String DOCUMENT_VERSION = "version";

	private Text societyName;

	private Text titleText;

	private Text versionText;

	/**
	 * Modify implementation of pxDoc Wizard
	 * ------------------------------------
	 */
	protected IPxDocWizard getPxDocWizard() {
		return (IPxDocWizard) getWizard();
	}

	@Override
	public void fillLauncher(PxDocLauncher pLauncher) {
		PxDocLauncherHelper.addProperty(pLauncher, SOCIETY, societyName.getText());
		PxDocLauncherHelper.addProperty(pLauncher, DOCUMENT_TITLE, titleText.getText());
		PxDocLauncherHelper.addProperty(pLauncher, DOCUMENT_VERSION,
versionText.getText());
	}

	public OptionPage(String pageName) {
		super(pageName);
		setTitle("Sample Option Page");
		setDescription("Illustrates how to implement a custom wizard page with generation
parameters.");
```



© EXTREME MODELING SOFTWARE 2018

```
}


/**
 * Design of the new Wizard page
 * -------------------------
 */


@Override
public void createControl(Composite parent) {
        Composite composite = new Composite(parent, SWT.NULL);
        GridLayout gridLayout = new GridLayout(2, false);
        composite.setLayout(gridLayout);

        // Document title
        Label titleLabel = new Label(composite, SWT.WRAP);
        titleLabel.setText("Document Title :");
        titleText = new Text(composite, SWT.BORDER);
        titleText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
        titleText.setMessage("A title for your document");

        // Document version
        Label versionLabel = new Label(composite, SWT.WRAP);
        versionLabel.setText("Document Version :");
        versionText = new Text(composite, SWT.BORDER);
        versionText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
        versionText.setMessage("1.0");

        // Society
        Label societyLabel = new Label(composite, SWT.WRAP);
        societyLabel.setText("Society :");
        societyName = new Text(composite, SWT.BORDER);
        societyName.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
        societyName.setMessage("Society Name");

        composite.pack();
        // set the composite as the control for this page
        setControl(composite);
}

@Override
public String getSectionId() {
```

```
		return SETTINGS_ID;
	}

	@Override
	public IDialogSettings buildDialogSettings() {
		DialogSettings settings = new DialogSettings(SETTINGS_ID);
		settings.put(SOCIETY, societyName.getText());
		settings.put(DOCUMENT_TITLE, titleText.getText());
		settings.put(DOCUMENT_VERSION, versionText.getText());
		return settings;
	}

	@Override
	public void loadDialogSettings(IDialogSettings pSettings) {
		if (pSettings != null) {

			// society
			String name = pSettings.get(SOCIETY);
			if (name == null) {
				name = "";
			}
			societyName.setText(name);
			// title
			String title = pSettings.get(DOCUMENT_TITLE);
			if (title == null) {
				title = "";
			}
			titleText.setText(title);
			// version
			String version = pSettings.get(DOCUMENT_VERSION);
			if (version == null) {
				version = "";
			}
			versionText.setText(version);
		}

	}

}
```

4. Use the properties in your Generator

In your generator, you can get the properties and use them as you want:

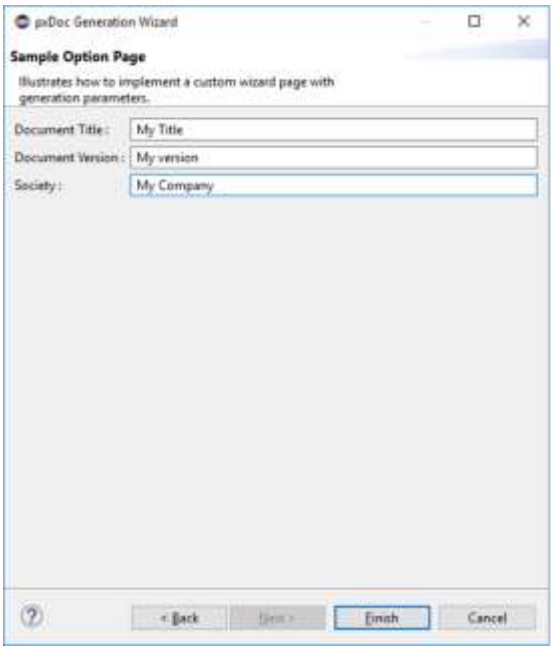


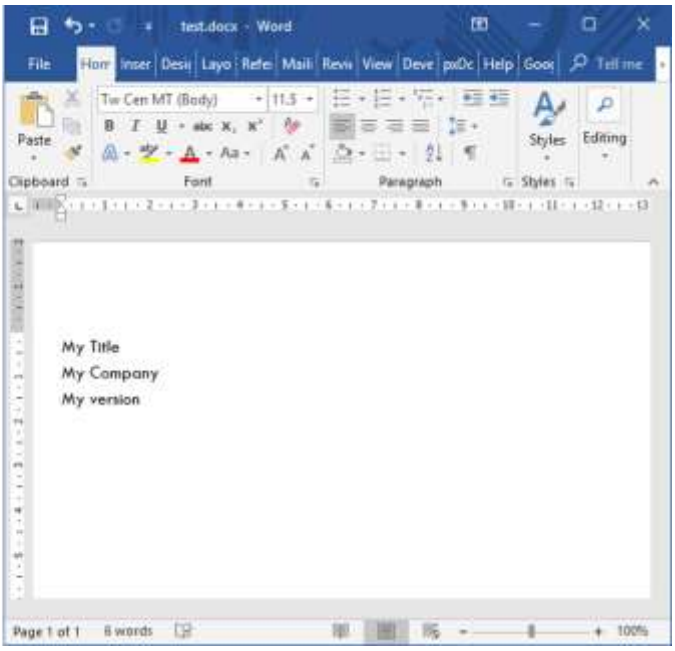
© EXTREME MODELING SOFTWARE 2018

```
package mygenerator

/**
 * MyGenerator empty pxDoc generator.
 */
pxDocGenerator MyGenerator
{

	root template main(Object model) {

		document {

			var specTitle = getProperty("title") as String
			var societyName = getProperty("society") as String
			var version = getProperty("version") as String

			specTitle
			§
			societyName
			§
			version

		}

	}

}
```

5. Result

Your new Wizard page is created and the properties are sent to the document:



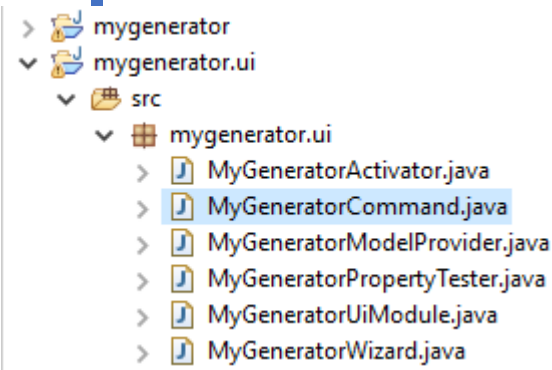




## 2. Bind variable styles before deploying a generator

### 1. Use case

The use case is the following: you have developed a Generator using variable styles, but you do not want your final user to have to choose a stylesheet nor to bind the styles before the generation. Before deploying your Generator (as a feature, plugin,...), you will just need to modify the code of the Command class as below.

> The Command class can be found in the *src* folder of the UI project linked to your generator.




Example with a generator named *MyGenerator* using the variable styles *MyStyle1* and *MyStyle2*: these styles will be bound to *Heading1* and *Heading2* of the deployed stylesheet *MyStylesheet*.



© EXTREME MODELING SOFTWARE 2018

```
package mygenerator.ui;

import com.google.inject.Inject;

import fr.pragmaticmodeling.pxdoc.runtime.IPxDocGenerator;
import fr.pragmaticmodeling.pxdoc.runtime.IStylesheetsRegistry;
import fr.pragmaticmodeling.pxdoc.runtime.launcher.PxDocLauncher;
import fr.pragmaticmodeling.pxdoc.runtime.util.PxDocLauncherHelper;

public class MyGeneratorCommand extends AbstractMyGeneratorCommand {

	@Inject
	IStylesheetsRegistry stylesheetRegistry;

	@Override
	protected void initLauncher(PxDocLauncher launcher) {
		super.initLauncher(launcher);
		IPxDocGenerator pxDocGenerator = getPxDocGenerator();
		pxDocGenerator.setStylesheet(stylesheetRegistry.getStylesheet("MyStylesheet"));
		PxDocLauncherHelper.addStyleBinding(pxDocGenerator, "MyStyle1", "Heading1");
		PxDocLauncherHelper.addStyleBinding(pxDocGenerator, "MyStyle2", "Heading2");
	}

}
```

## 2. pxDoc - Common Libraries and Plugins

pxDoc Common Libraries and Plugins are a set of pxDoc Modules and/or Java classes or interfaces providing useful extensions capabilities, that can be reused in any pxDoc Generator. Among others, you can:

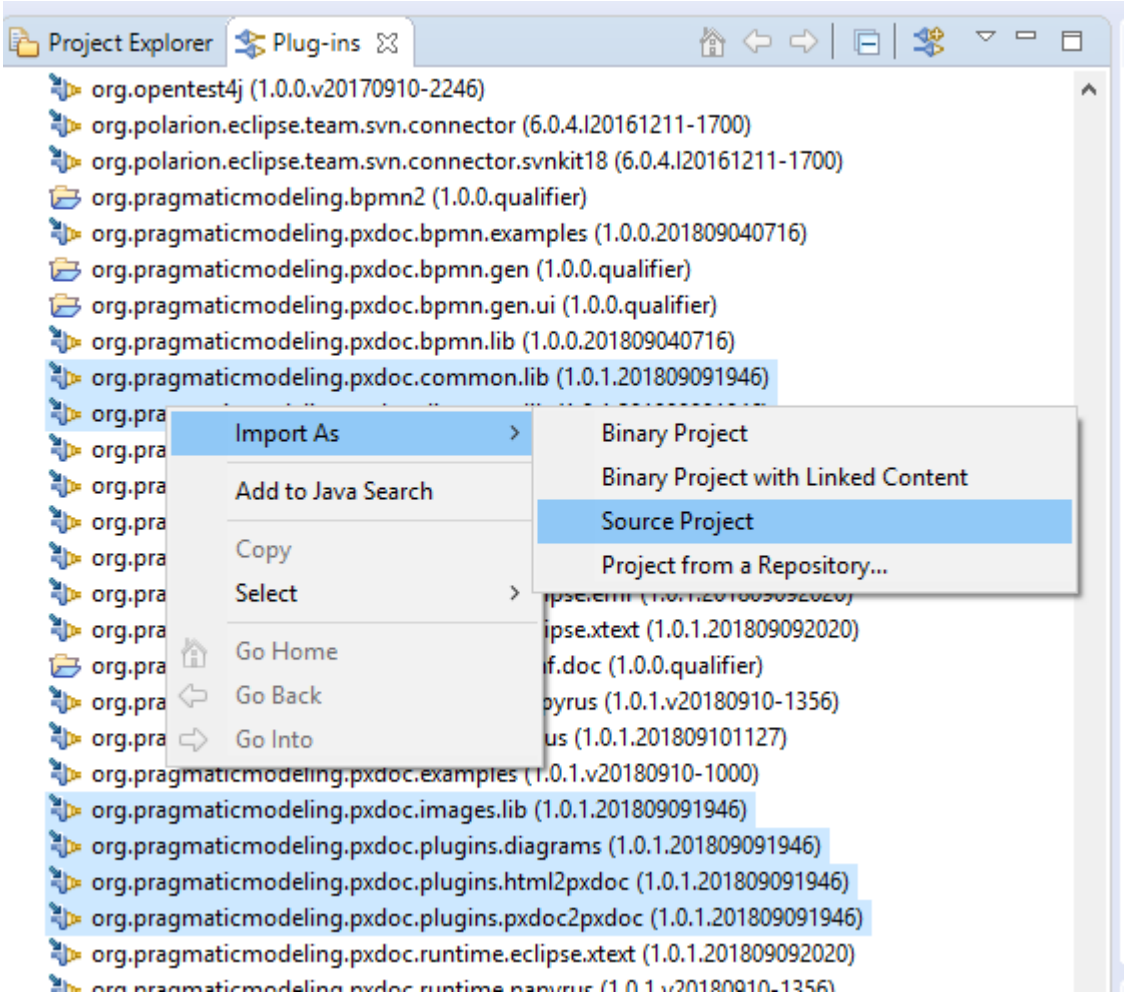
- query and include diagrams from your models,
- provide generic templates for EMF model,
- manipulate images (convert to AWT, split large images on several pages...)
- ...

The language renderers are part of the pxDoc common plugins but they are described in a dedicated section.

> There is no obligation to use these common libraries. You can also develop your own libraries, or extend the provided ones to match your specific needs.

## 1. Importing sources into your workspace

You can of course import the common libraries in your workspace to get a view on the common Modules and libraries included in pxDoc




## 2. Common Services / *org.pragmaticmodeling.pxdoc.common.lib*

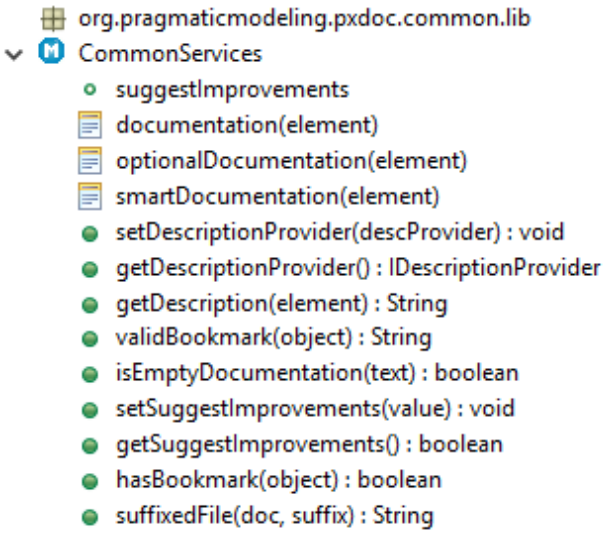
> The Common Services library is provided for demonstration purpose, and as a support for the set of generator examples shipped with pxDoc. You are free to use or not this library, or find inspiration to define your own *"common lib"*.


© EXTREME MODELING SOFTWARE 2018

## 1. CommonServices.pxdoc Module

The *CommonServices.pxdoc* module provides generic templates and functions that can be used by every documentation generator:

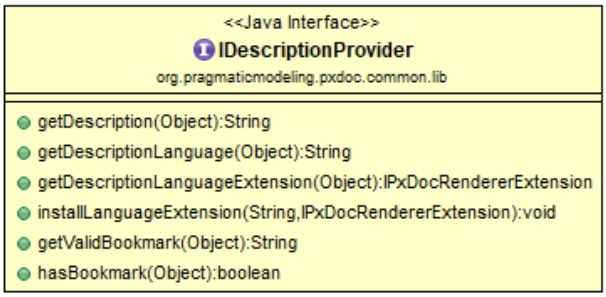
- manage basic presentation of an object (so far, bookmarks management and objects descriptions retrieval). This is useful if your generator consumes many different java objects, with different ways of storing documentation, occasionally in many different documentation formats (raw text, html, markdown...)
- provide basic reusable templates for optional or required documentation
- provide a way to improve your documents, with the *suggestImprovements* boolean parameter. When set to true, warnings are generated in the document to enforce writing a documentation for the concerned objects.
- ... and more useful generic services to come in the next releases!




> CommonServices setup
>
> **OPTIONAL** - When using the CommonServices module in your generator, you can inject your custom instance of IDescriptionProvider (see example below).

### 2. The *IDescriptionProvider* Interface

Among all properties that an object can hold, its human readable descriptions play a particular role. They are usually generated early when documenting an object, while other business properties are treated differently in following document sections.




The *IDescriptionProvider* interface gives the ability to return a description for every object processed by a pxDoc generator, as well as the expected description format (e.g., html, raw text, markdown...)

It is also responsible to provide a valid bookmark id for every object processed during generation, and decide which objects should be generated with a bookmark.

The class *DefaultDescriptionProvider* is the default implementation of *IDescriptionProvider*. It returns null as a description for every object, but is able to generate valid bookmarks for every object.

If you plan to use the CommonServices module, you will need to :

- provide your own *IDescriptionProvider* by subclassing *DefaultDescriptionProvider* and provide your own implementations of the getDescription*(*) methods.
- inject your IDescriptionProvider instance to the CommonServices when starting the generation (in the root template):

```
root template main(Object model) {

	// Commons lib configuration : set a description provider able to get information about object description
	descriptionProvider = new MyDescriptionProvider(this)

	document {
		// content...
	}

}
```

## 3. EMF Services / *org.pragmaticmodeling.pxdoc.emf.lib*


© EXTREME MODELING SOFTWARE 2018

> The Emf Services library is provided for demonstration purpose, and as a support for the set of generator examples shipped with pxDoc. You are free to use or not this library, or find inspiration to define your own *"emf lib"*.

The EmfServices module is a set of templates and functions that allow to generate documentation for any EMF EObject. It takes benefit of the reflective EMF API to transform EObjects into documentation.

It defines two ways of EObject-to-doc transformation:

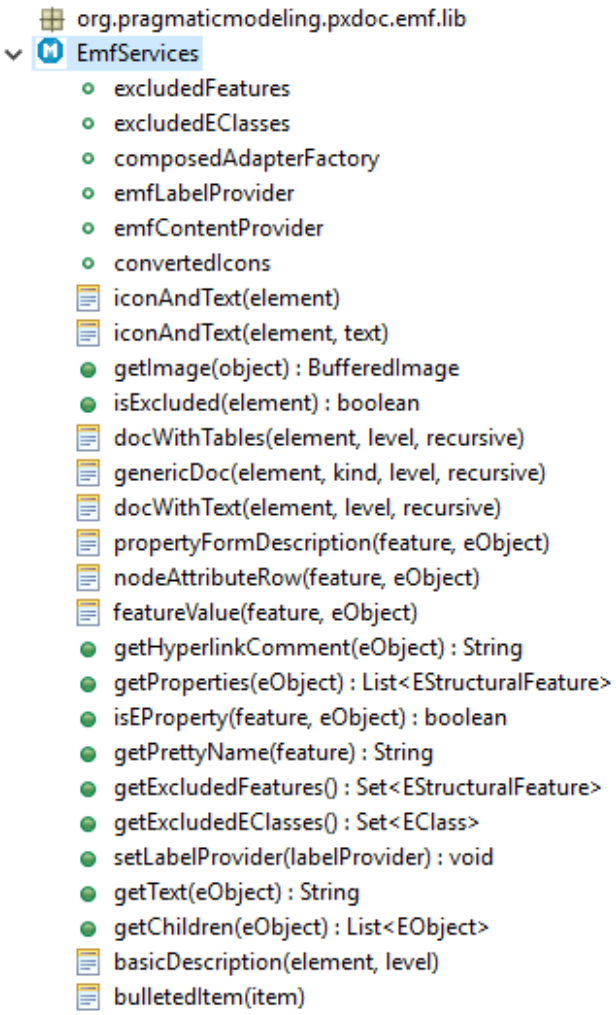
- *DocKind.form*, for a form-like presentation of EObject features (attributes, references, methods)
- *DocKind.table*, for a table regrouping all EObject features

You can get your documentation for an EObject by invoking the *genericDoc* template:

```
var int level = 1
var boolean recursive = true
apply genericDoc(anEObject, DocKind.^table, level, recursive)
```

With:

- *anEObject*, the model object
- *docKind*, between *form* or *table*
- *level* gives the heading (title) level that should be applied to the model object
- *recursive*, to generate the documentation recursively on the object's children




And because some EObject's attributes are too technical, or rarely/never used, the EmfServices module provide a way to filter some EStructuralFeature from the generation. This is the role of the field *excludedFeatures*, which you can access the following way:

```
root template main(Object model) {

	// exclude some EClasses
	excludedEClasses += EcorePackage.Literals.EANNOTATION
```

```
        // exclude EStructualFeatures
        excludedFeatures += EcorePackage.Literals.ECLASSIFIER__DEFAULT_VALUE
        excludedFeatures += EcorePackage.Literals.EMODEL_ELEMENT__EANNOTATIONS

        document {
                // content...
        }

}
```

The EmfServices.pxdoc files contains comments to explain each template. Do not hesitate to have a look at it.

You can also refer to the EMF Example to see how these templates can be used.

EmfServices setup

**OPTIONAL** - When using the EmfServices module in your generator, you can inject your custom instances of:

- labelProvider = myLabelProvider
- contentProvider = myContentProvider

**OPTIONAL** - You can also define filters to exclude some model elements or features from the generation, by adding EClass or EStructuralFeature to the two following sets:

- excludedEClasses += MyEmfPackage.Literals.MY_CLASS
- excludedFeatures += MyEmfPackage.Literals.MY_CLASS__FEATURE

## 4. Diagram Services / *org.pragmaticmodeling.pxdoc.diagrams.lib*

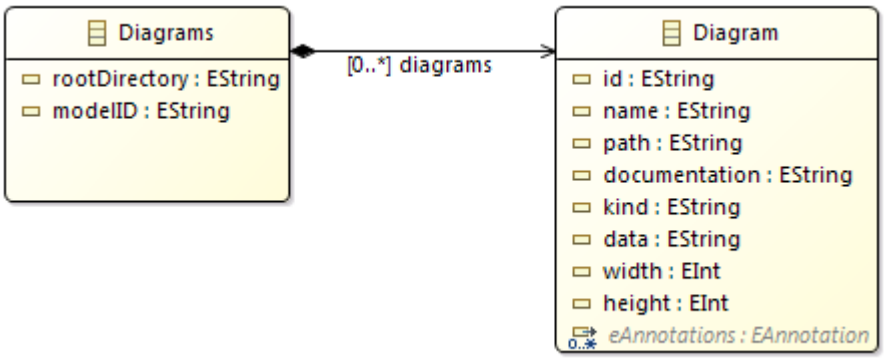
> The Diagram Services library is provided for demonstration purpose, and as a support for the set of generator examples shipped with pxDoc. You are free to use or not this library, or find inspiration to define your own *"diagrams lib"*.

### 1. pxDoc Diagrams metamodel

A very simple EMF metamodel has been defined to manipulate diagrams in pxDoc. It is composed of a root object *Diagrams*, which contains a list of *Diagram*.



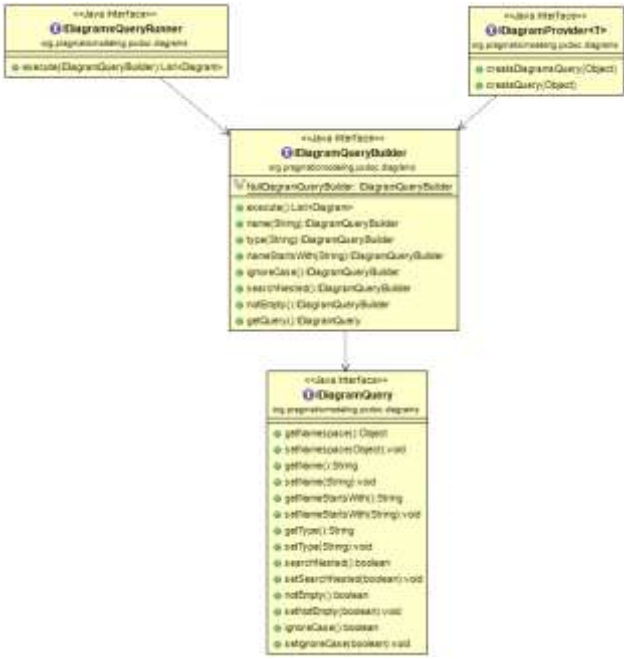
© EXTREME MODELING SOFTWARE 2018




## 2. pxDoc Diagrams runtime

The diagrams library provides a set of interfaces to query diagrams from an object.

The diagrams library provides a set of interfaces to query diagrams from an object:



- **IDiagramProvider**: this is the entry point to query diagrams, by invoking its *createQuery(modelObject)* method, which returns an *IDiagramQueryBuilder* object.
- **IDiagramQueryBuilder**: defines a set of simple queries (by name, by type, by non empty diagrams...), that can be stacked to the query. When the query is complete, the *execute()* method has to be called to run the query. For example, there are 2 unit queries in the following query: *diagramProvider.createQuery(sourceObjectWithDiagrams)**.type("CDB").notEmpty()**.execute*.
- **IDiagramQuery**: a POJO for the diagram query
- **IDiagramQueryRunner**: will run the *IDiagramQuery*

> The Diagram library provides useful templates to include diagrams.
>
> You can also find samples in the examples provided with the pxDoc enablers

### 3. DiagramServices.pxdoc Module

#### Diagram queries with *queryDiagrams(Object)*

Samples of queries:

```
// diagrams owned by the 'model' object
var List<Diagram> ownedDiagrams = queryDiagrams(model).execute;

// all diagrams owned directly and indirectly by the 'model' object
var allOwnedDiagrams = queryDiagrams(model).searchNested.execute;

// diagrams owned by the 'model' object, with some content
var ownedNonEmptyDiagrams = queryDiagrams(model).notEmpty.execute;

// look for a diagram owned by 'model' and named 'A diagram name'
var diagramsByName = queryDiagrams(model).name('A diagram name').execute;

// look for a diagram owned by 'model' with name staring with 'ABC'
var diagramsByNameStartingWith = queryDiagrams(model).nameStartsWith('ABC').execute;

// diagrams by type
var diagramsByType_CDB = queryDiagrams(model).type('CDB').execute;
var diagramsByType_ES = queryDiagrams(model).type('ES').execute;
var diagramsByType_CDI = queryDiagrams(model).type('CDI').execute;

// diagrams with type 'CDB' and name starting with 'MyString'
var cdbStartingWithMyString = queryDiagrams(model).type('CDB').nameStartsWith('MyString').execute;
```

#### Include diagrams with the includeDiagram(Diagram) template

You can use the includeDiagram template to include your diagrams:

```
// To insert the diagram image by yourself:
for (diagram : myDiagrams) {

        // use the pxdoc 'image' keyword with diagram image file path
        // stored in the path diagram attribute  (was 'data' in pxDoc v1.1 and previous).
        image path:diagram.path
        // add a title with a style
        #Figure keepWithNext align:center {
        diagram.name
        }
        // include diagram documentation if any
        if (!diagram.documentation.empty) {
```



© EXTREME MODELING SOFTWARE 2018

```
                #BodyText {diagram.documentation}
        }
}
```

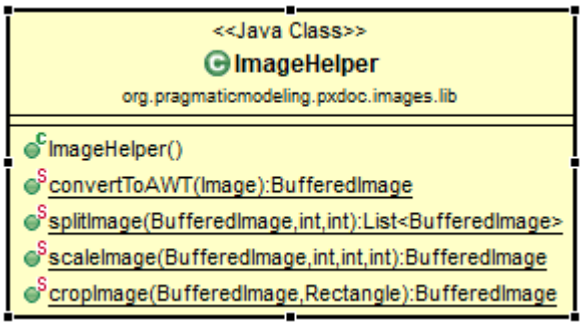
> DiagramServices setup
>
> **REQUIRED** - When using the DiagramServices module in your generator, you must inject an *IDiagramProvider* suited to your target platform.
>
> diagramProvider = myToolDiagramProvider

## 5. Images Services / *org.pragmaticmodeling.pxdoc.images.lib*

> The Images Services library is provided for demonstration purpose, and as a support for the set of generator examples shipped with pxDoc. You are free to use or not this library, or find inspiration to define your own *"images lib"*.

Set of function allowing to:

- Split large images in to several ones (and calculate automatically if split is necessary)
- Crop an image, to display only a part of it
- Scale an image to keep its aspect ratio
- Convert a Java Image object into an AWT object (pxDoc can manipulate image files or AWT objects)



# 6. Install pxDoc License

## 1. Evaluation License

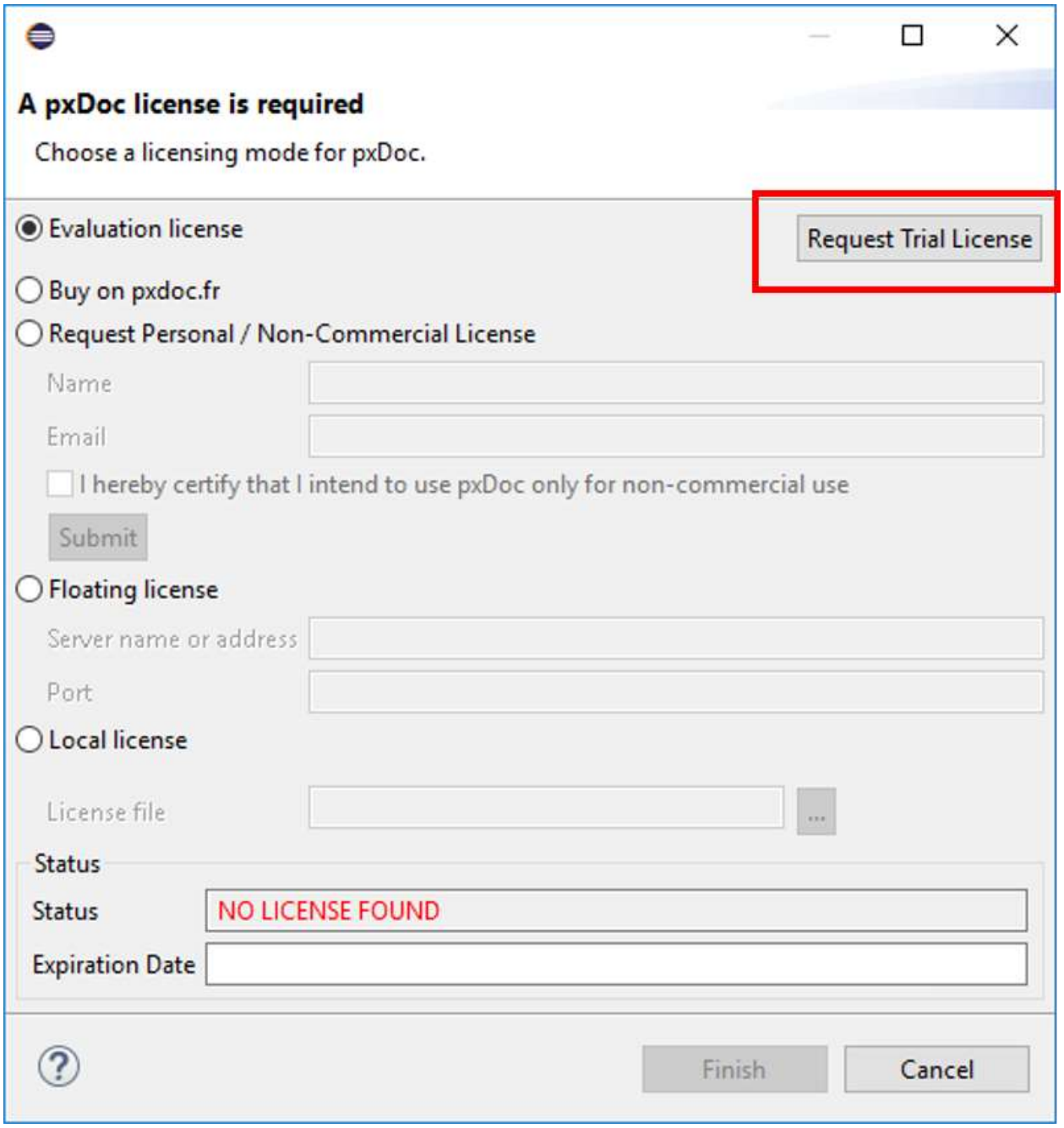
pxDoc is provided with an evaluation license **valid for 30 days**.

After this period, you will have to get a Personal license (non-commercial use only) or to purchase fixed or floating licenses of pxDoc on in online store.

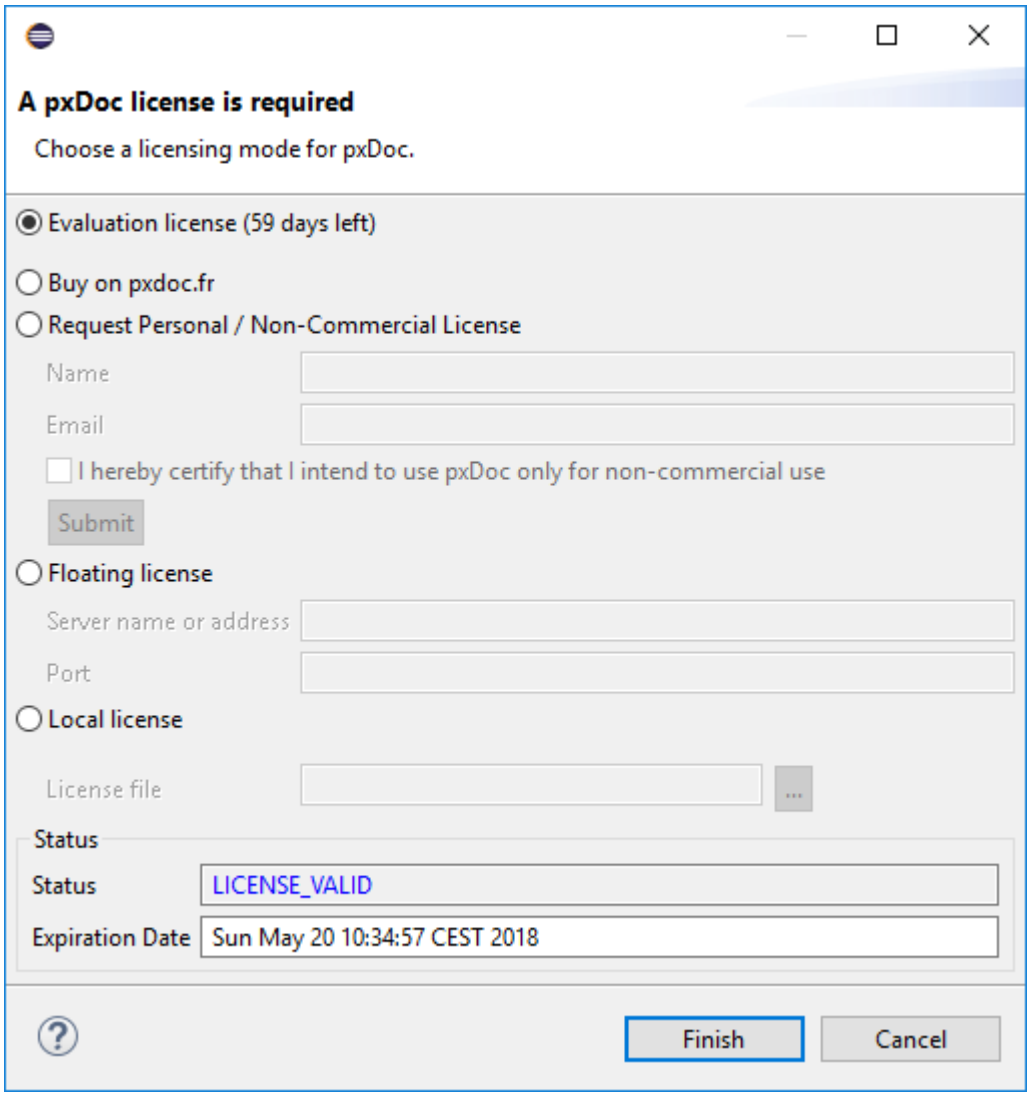
The first time you launch a document generation, a pop-up will appear requesting a valid license: to try pxDoc, click on *Request Trial License*:



© EXTREME MODELING SOFTWARE 2018




The evaluation license is activated and the trial period starts. Each time you will launch a document generation, the pop-up will appear, showing the remaining days.



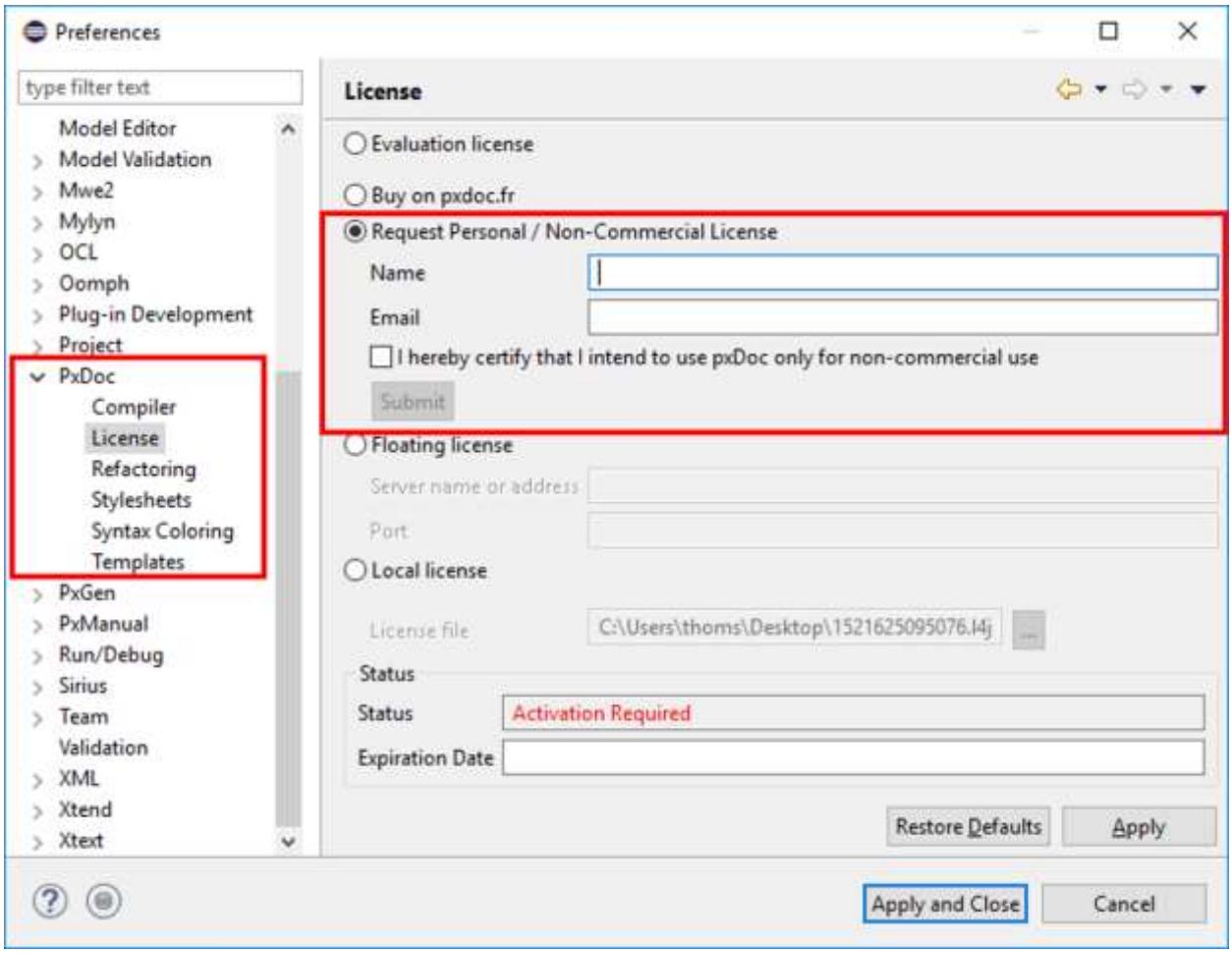

## 2. Personal License

If your activity is not commercial (personal, educational, non-profit organization), you can request a Personal License, which is **free of charge**.

This can be done either through the pop-up window that appears when you have an invalid or evaluation license, or via the Eclipse Preferences:



© EXTREME MODELING SOFTWARE 2018




Enter your name and a valid email address: your license file will be sent by email. You need to certify that you intend to use pxDoc only for non-commercial use (check the box) before submitting your request.

Once you have received your license, insert it as a *Local License*: follow the same steps as for the Fixed License.

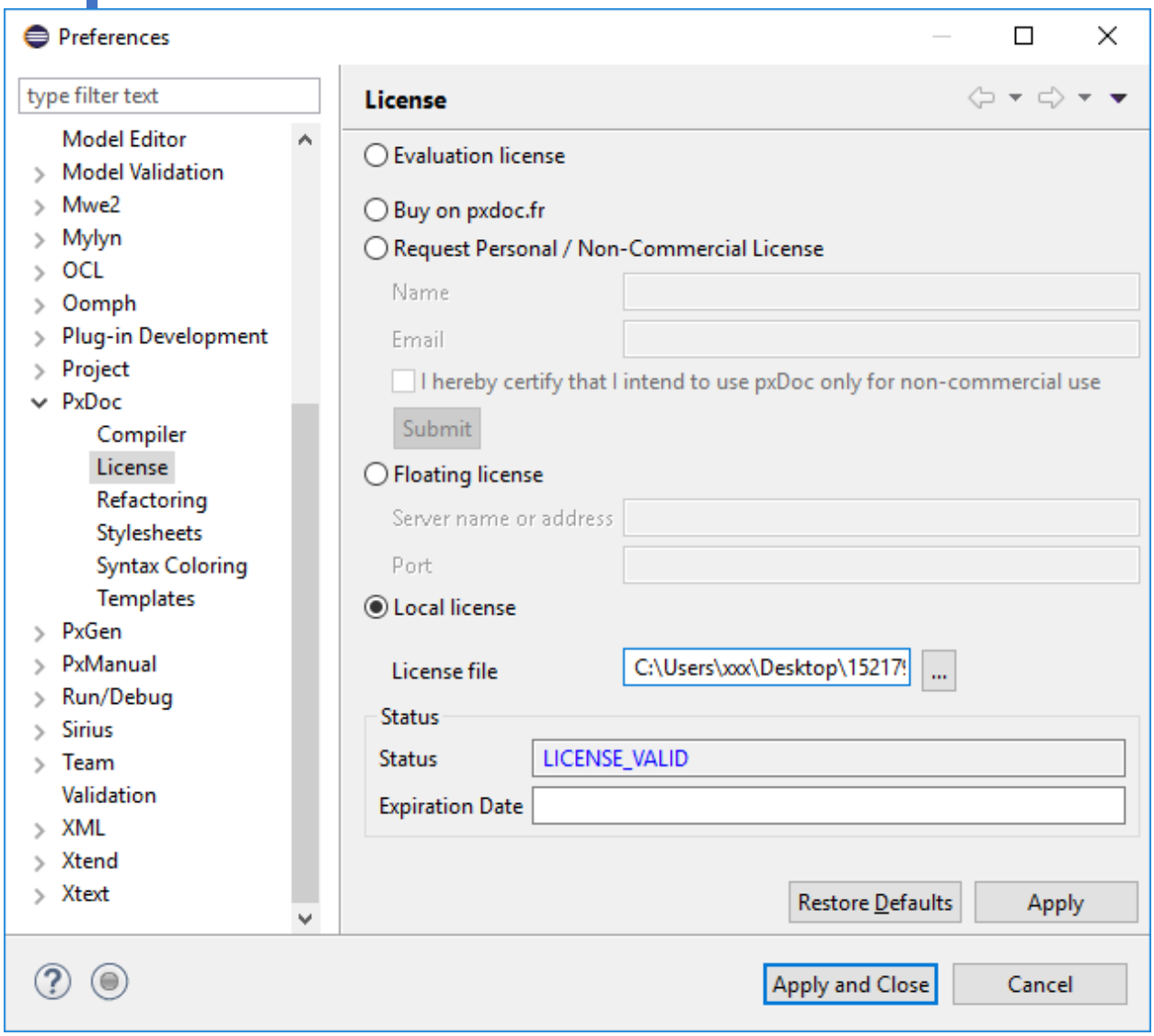
## 3. Fixed License

In pxDoc preferences license section, click on *Local license* and select the license file you received by email.

Your license will be automatically activated, if you are connected to the Internet

> License activation requires an Internet connection. If you are in a secured environment and connection fails, please contact your IT administrators.

> We can also provide a proxy server in case you need it. Do not hesitate to contact us: support@pxdoc.fr.




> The activated license file is sent to you by email for backup purposes only as it is automatically installed in pxDoc during activation.

**After activation, your license is linked to the "Hardware ID" of your device.** If some characteristics of your device change, your license will no longer be valid.

Please contact us if you need to update your license activation with a new Hardware ID: support@pxdoc.fr



© EXTREME MODELING SOFTWARE 2018

# 4. Floating License

## 1. Install Floating License Server and get its Hardware ID

To get your License, you will need to send us the "Hardware ID" of the server where the Floating License Server is installed.

Please follow the instructions in the document below to download and install the License4J Floating License Server (and Manager) on your network environment:

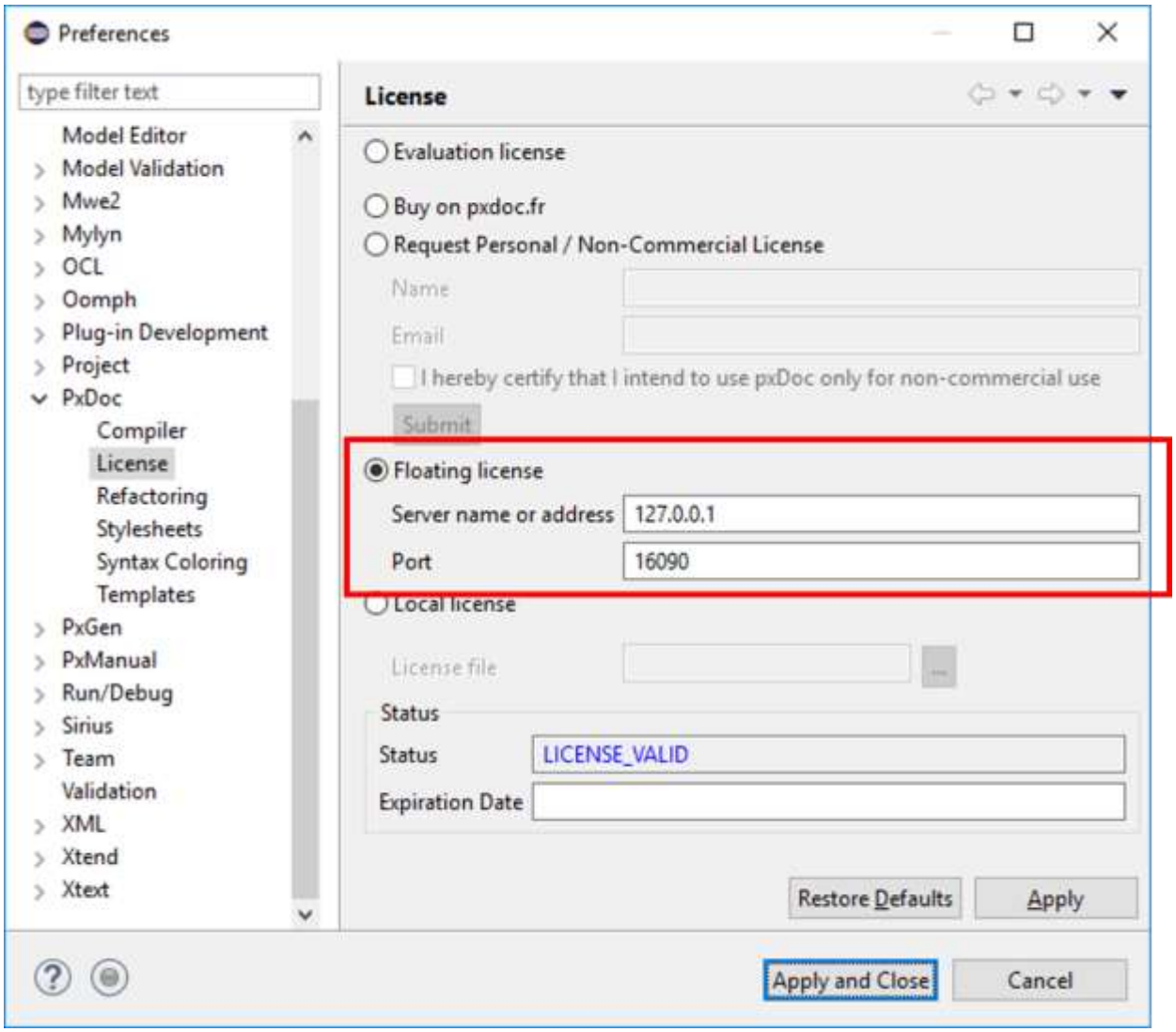
- **pxDoc Floating License Server Installation Guide**

Refer to *section 3.2 "Displaying License Server Hardware IDs"* to get your Hardware ID and send it back with your Order Confirmation at support@pxdoc.fr.

Once you have received the license file, follow the instructions in the installation guide to install it via the Floating License Server Manager.

## 2. Specify Floating License Server location in pxDoc

In pxDoc preferences license section, click on *Floating license* and enter the relevant IP address and port (default: 16090).






© EXTREME MODELING SOFTWARE 2018