Acceleo

Table of Content

1. Introduction

Acceleo 4 is a a text generator based on templates. It can be used to generate any kind of text file: code, configuration, documentation,...

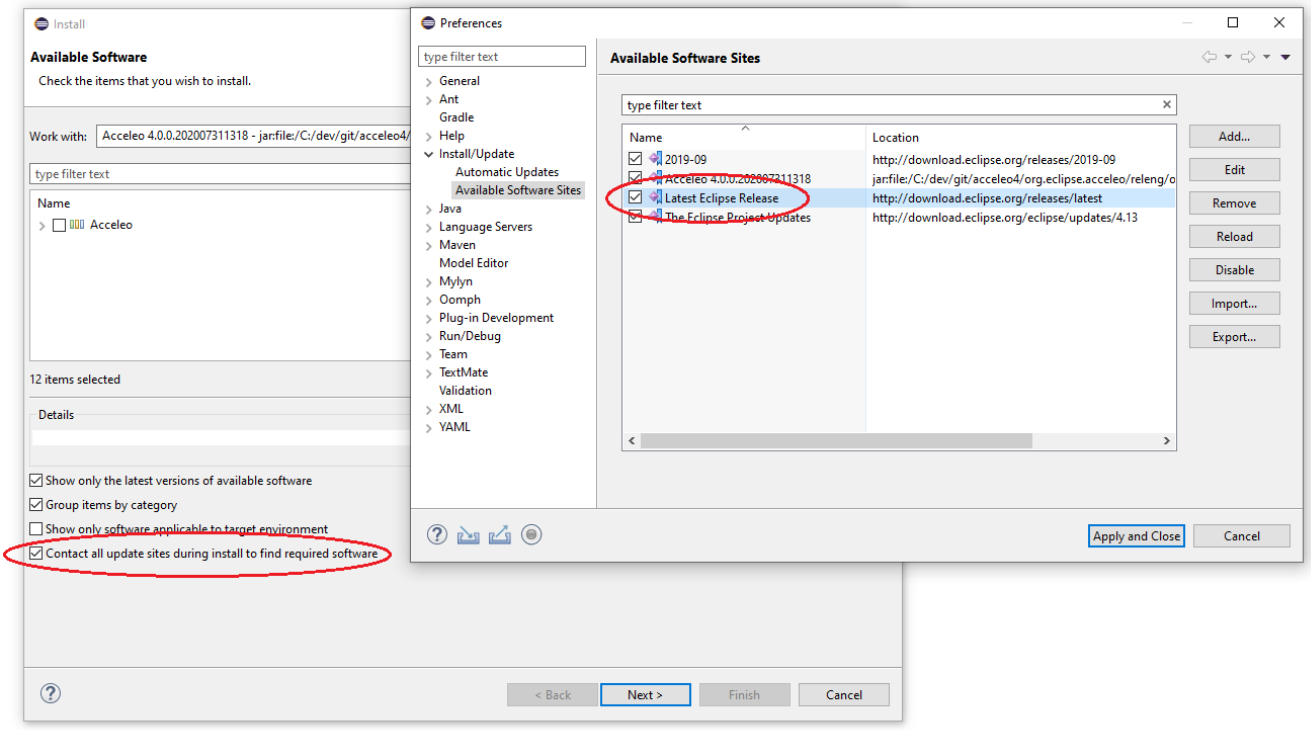
The template consists of imperative statements like conditionals, loops, and navigation expression used to retrieve data from models.

When generating, the engine use templates statements and also replace navigation expressions by their values in order to produce the output text.

2. Installation

If you have an existing Eclipse installation and simply wish to install Acceleo in it, you can install it from the **Update Site**, available here https://download.eclipse.org/acceleo/updates/nightly/.

When installing the update site you must ensure that **"Contact all update sites during install to find required software"** is checked and that in the **"Manage..."** dialog, the **"Latest Eclipse Release"** update site is checked.



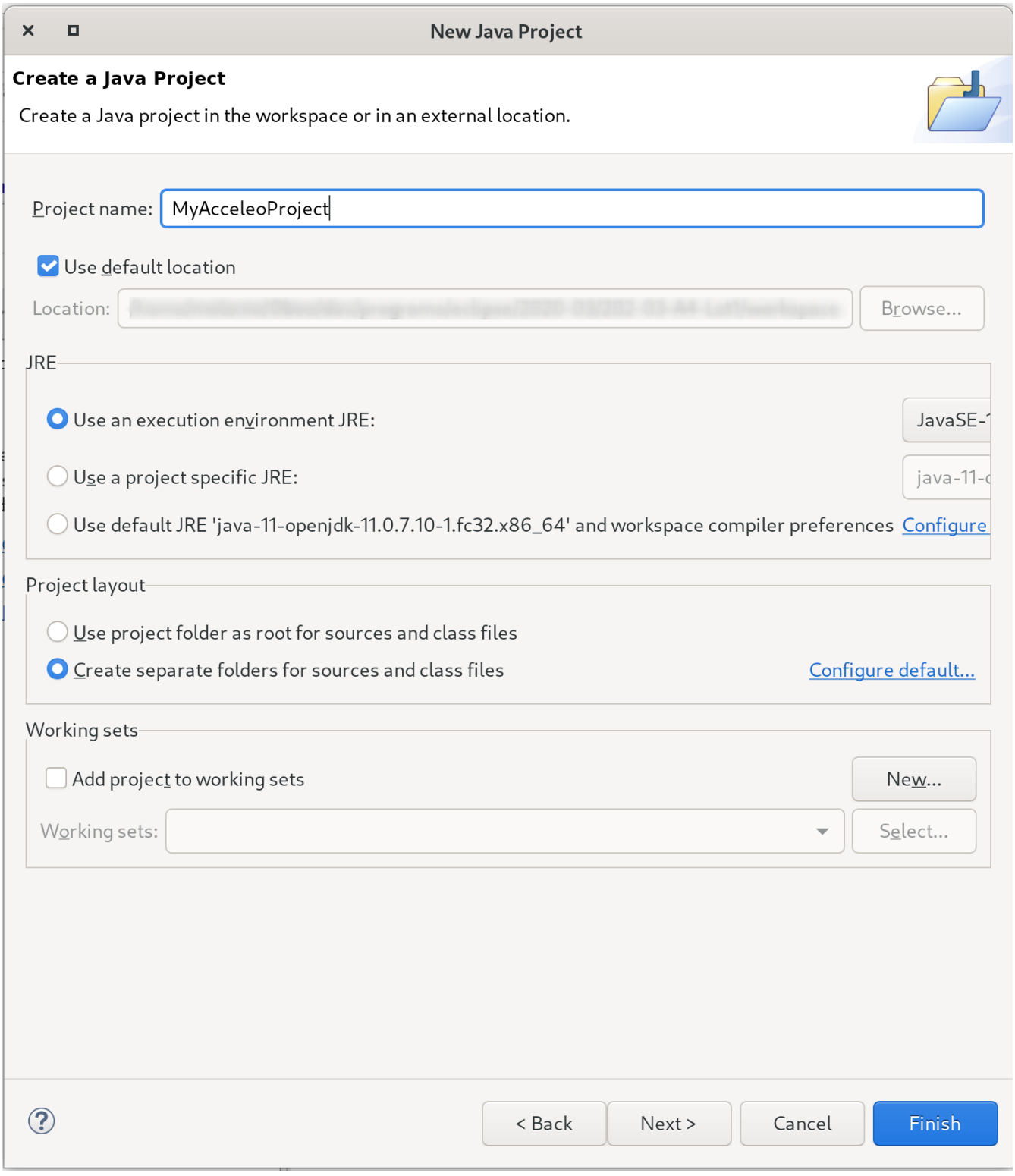
For those of you who need to retrieve the source code of Acceleo, it is available on Git: https://git.eclipse.org/c/acceleo/org.eclipse.acceleo.git/.

3. Getting Started

In this section, you will create a new project to work with Acceleo.

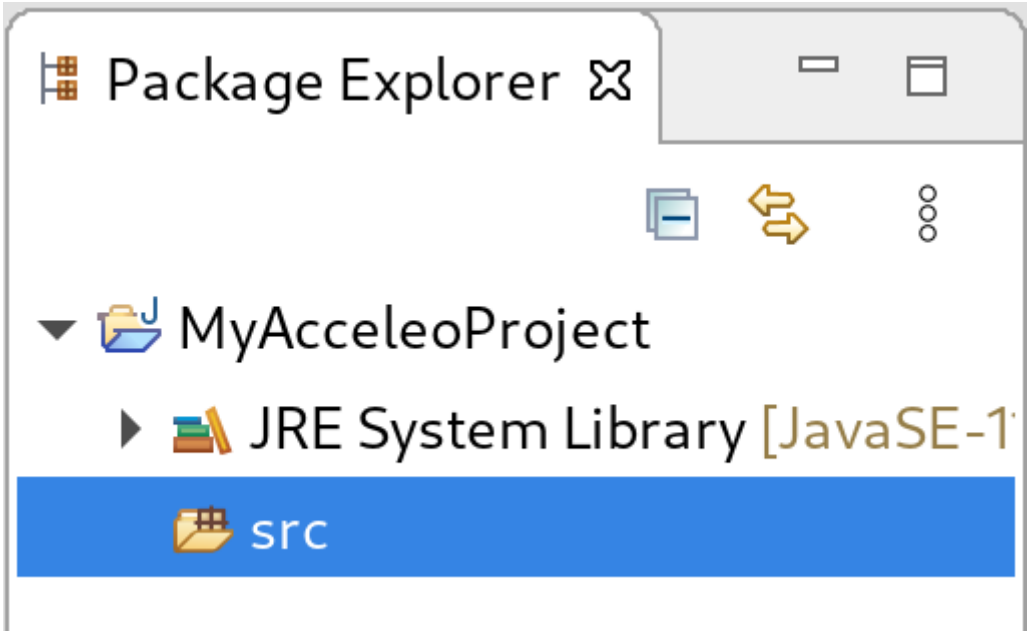
First you need to create a **Java project**:

1. Inside Eclipse select the menu item **File** > **New** > **Project...** to open the **New Project** wizard,
2. Select **Java Project** then click **Next** to start the **New Java Project** wizard:

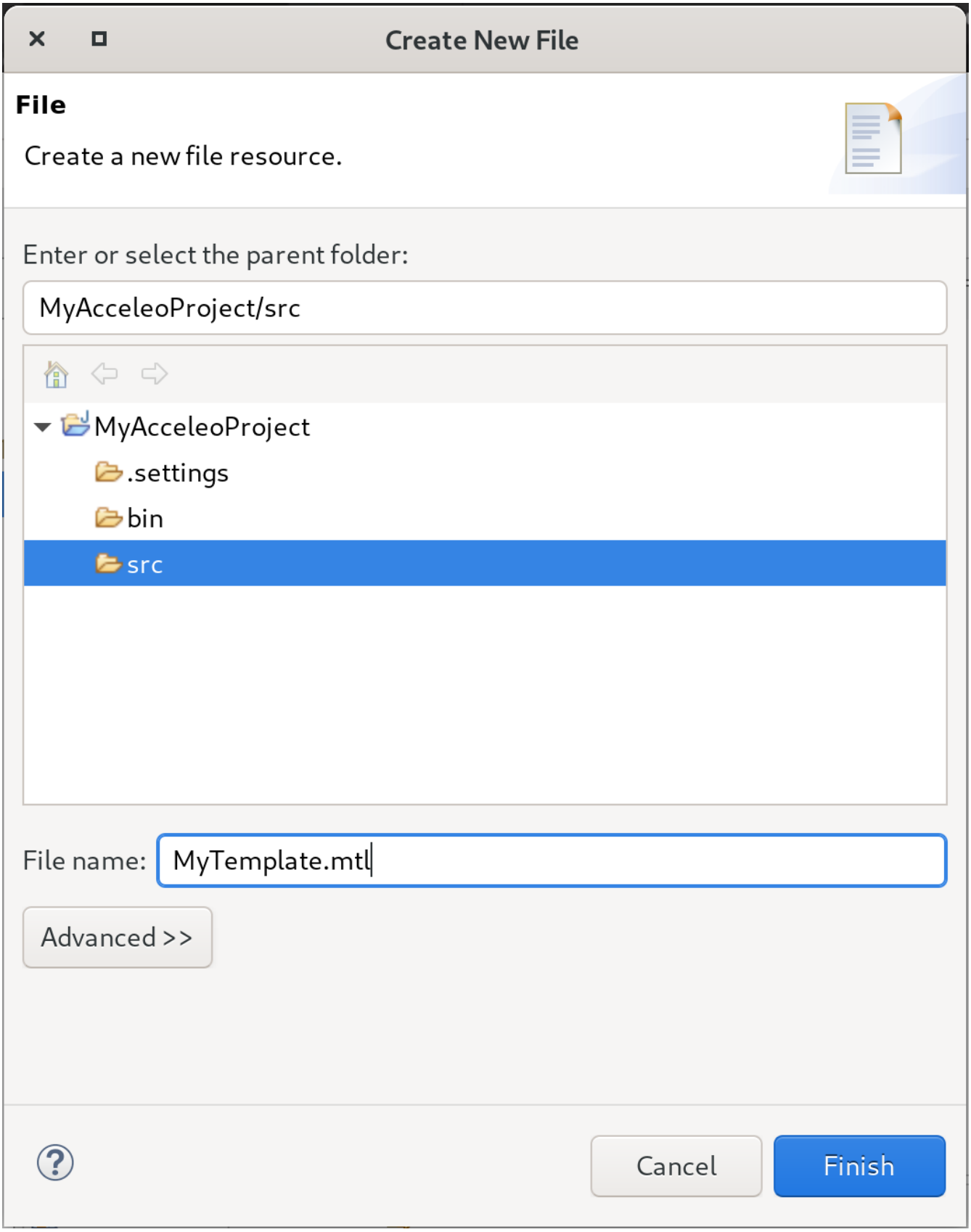


On this page:

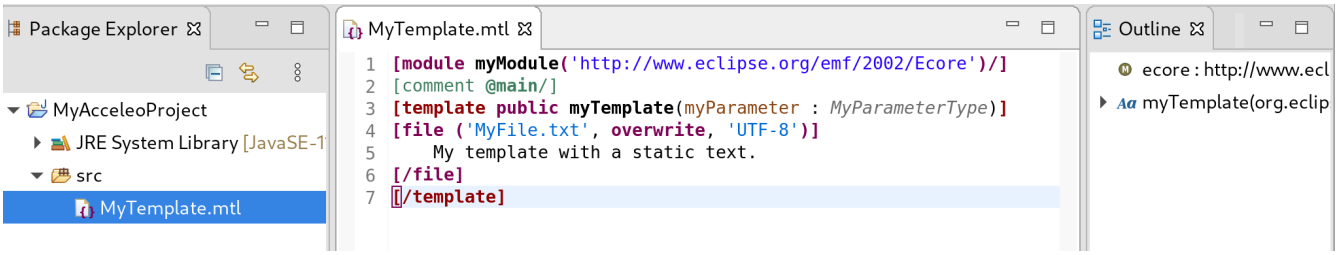
1. Enter the Project name,
2. Select the Java Runtime Environment (JRE) or leave it at the default,
3. Then click **Finish**.



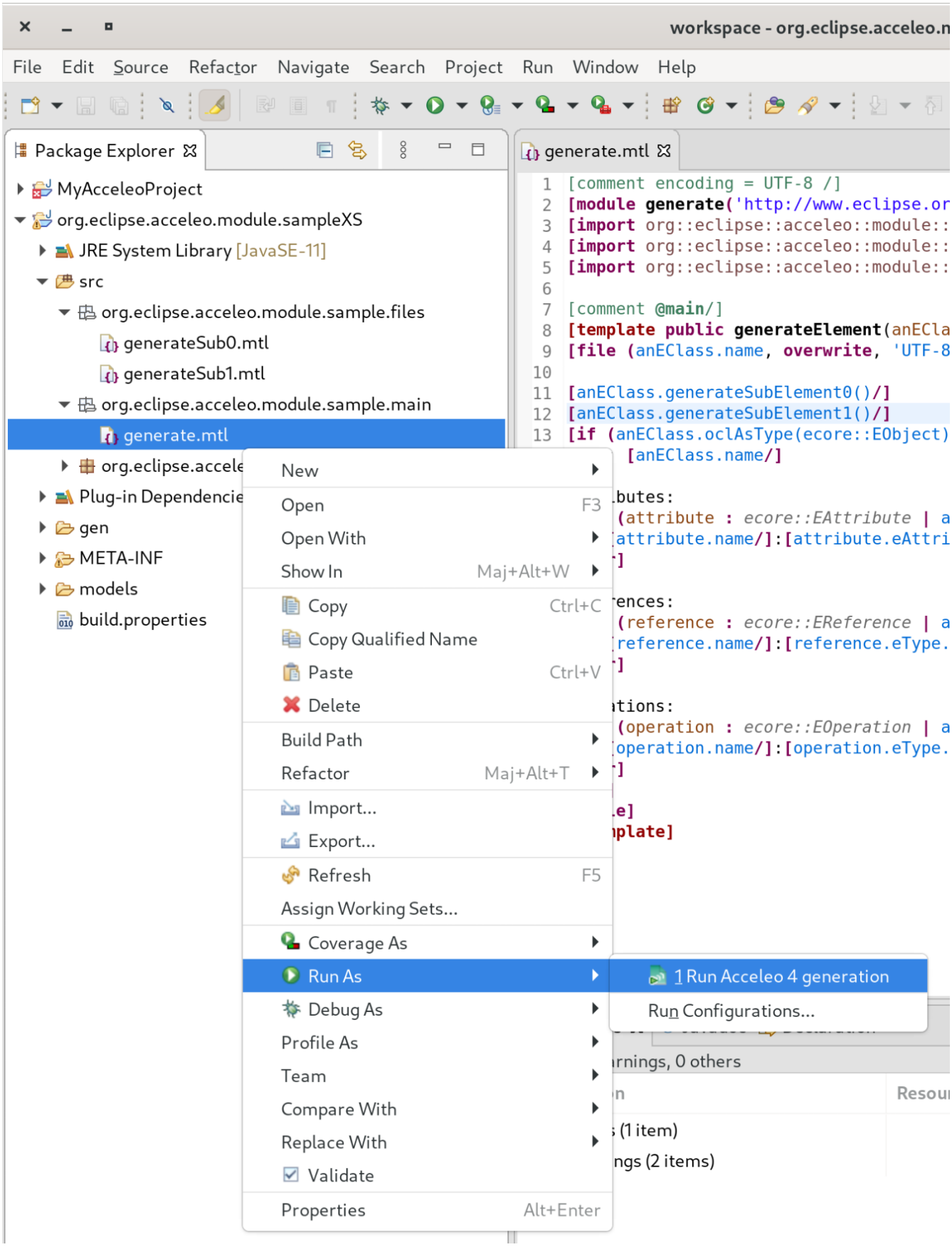
1. In the **Package Explorer**, expand the project and select the source folder **src**,
2. Select the menu item **File** > **New** > **File** to open the **New File** wizard,
3. Select the **src** folder as parent folder,
4. Enter the File name with the extension **.mtl**, for example: `MyTemplate.mtl`,
5. Then click **Finish**.



An empty editor opens, you can start writing your generation template.



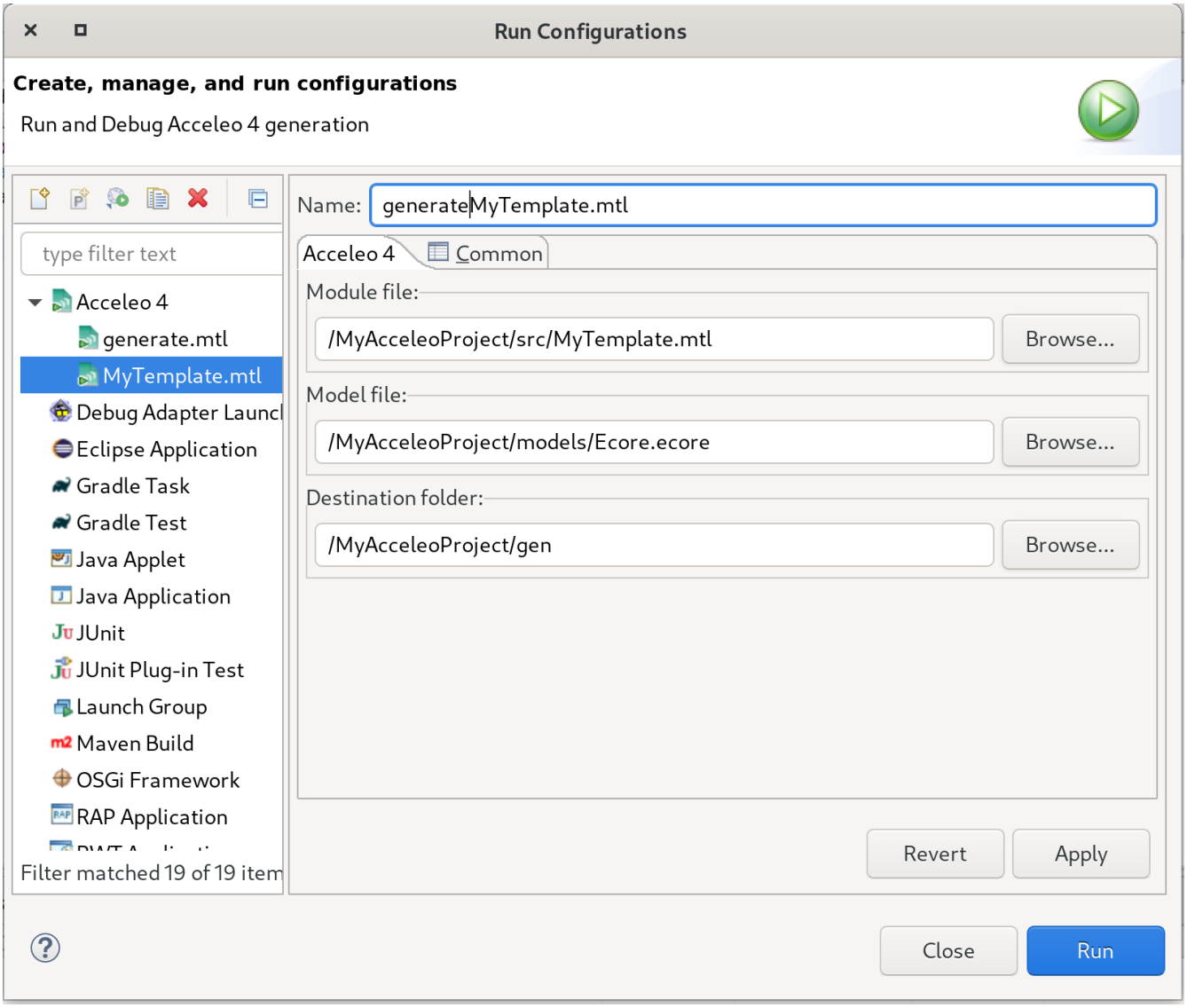
In order to launch an Acceleo generator, you just have to right-click on your main module and use the **Run As** menu.



From there the **Launch Configuration** menu will open. If you already have created a launch configuration you can access the launch configuration menu from the **Run** > **Run Configurations...** menu. In this menu, you will have access to the Acceleo Launch configuration. You just have to select:

- your main **module file**,

- your input **model file**,
- the **destination folder** of the generation.



INFO: Check that your generation is finished by opening the **Console** view. You should see a message like this `<terminated> generate.mtl[Acceleo4]`.

4. The Acceleo Tooling

4.1. The Acceleo Editor

Before you Start

Use a **Java Project** and the classical **Java Perspective** for writing your Acceleo templates `.mtl` in the **src** folder.

The module editor provides the following features:

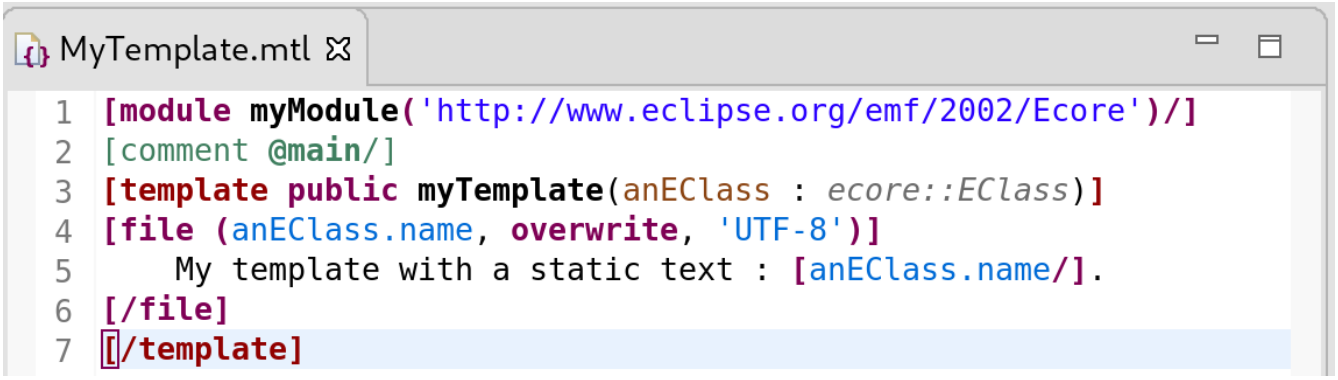
- Syntax highlighting;
- Content assistant (`Ctrl + Space`);

- Error detection;
- Outlines;
- Quick outline (`Ctrl + O`);
- Open declaration (either with `Ctrl + Left Click` or `F3`),
- White spaces.

4.1.1. Syntax Highlighting

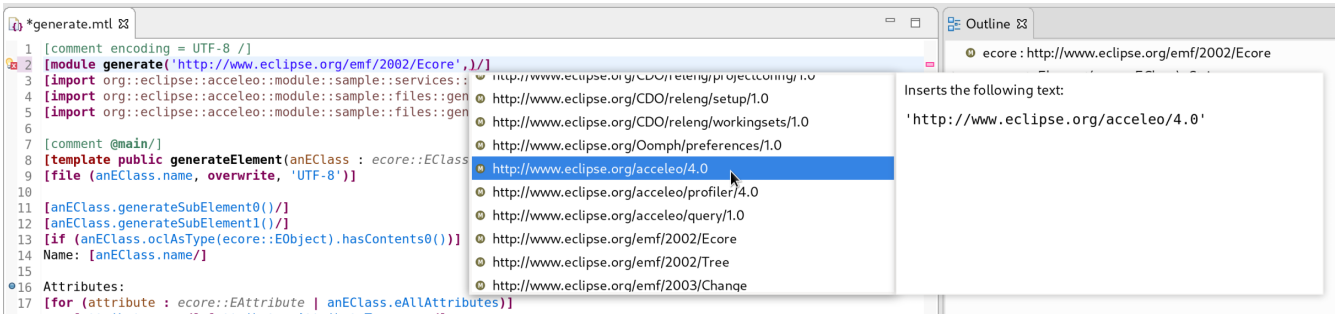
The editor uses specific colors for Acceleo templates:

- red is used for template tags;
- purple is used for other tags (queries, modules, imports, ...);
- blue is used for dynamic expressions in templates or other places;
- green is used for comments and String literals;
- black is used for static text or query bodies.



4.1.2. Content assist

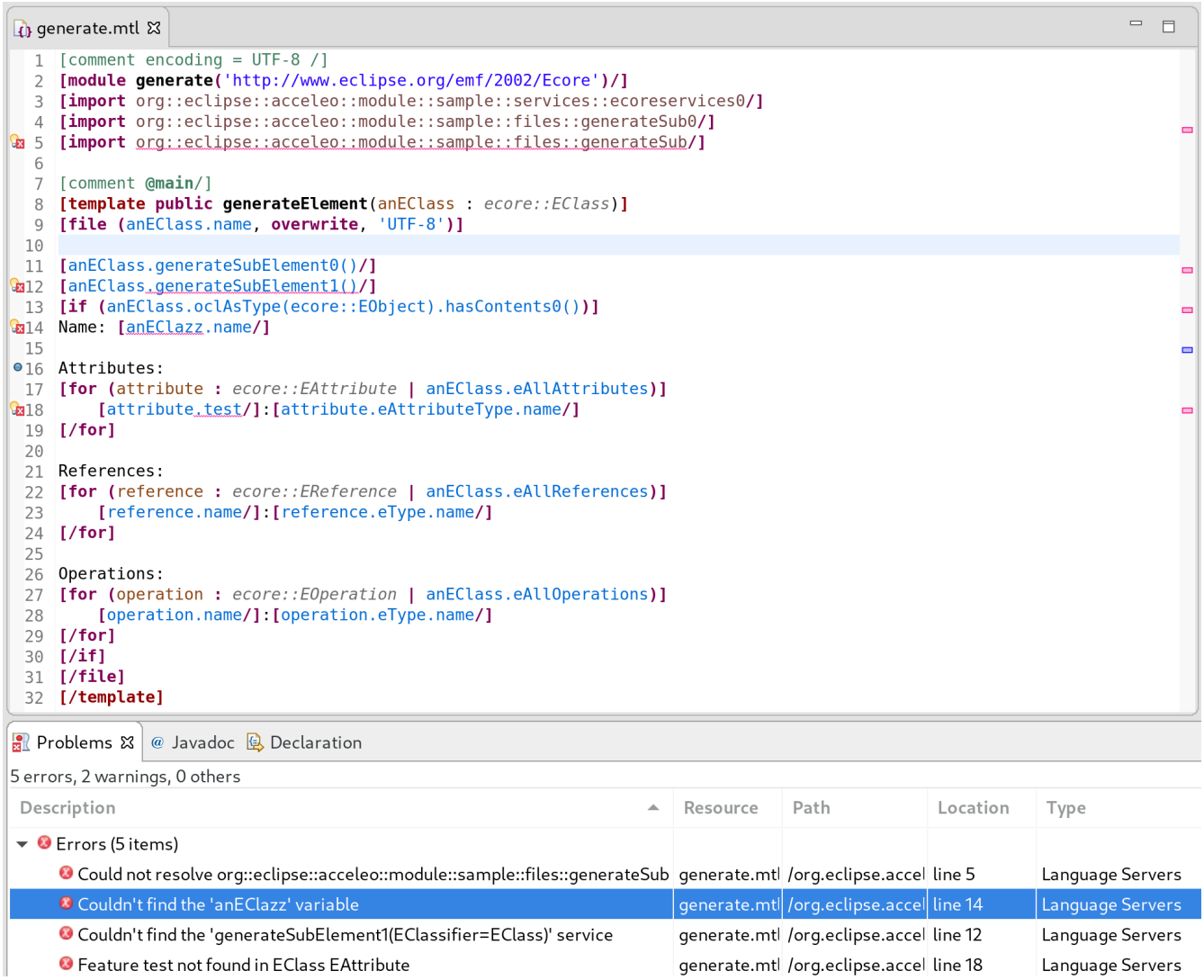
The content assistant is traditionally invoked with `Ctrl + Space`. It proposes a choice of all elements that make sense at the place of invocation. It is available everywhere, so don't hesitate to hit `Ctrl + Space` anywhere!



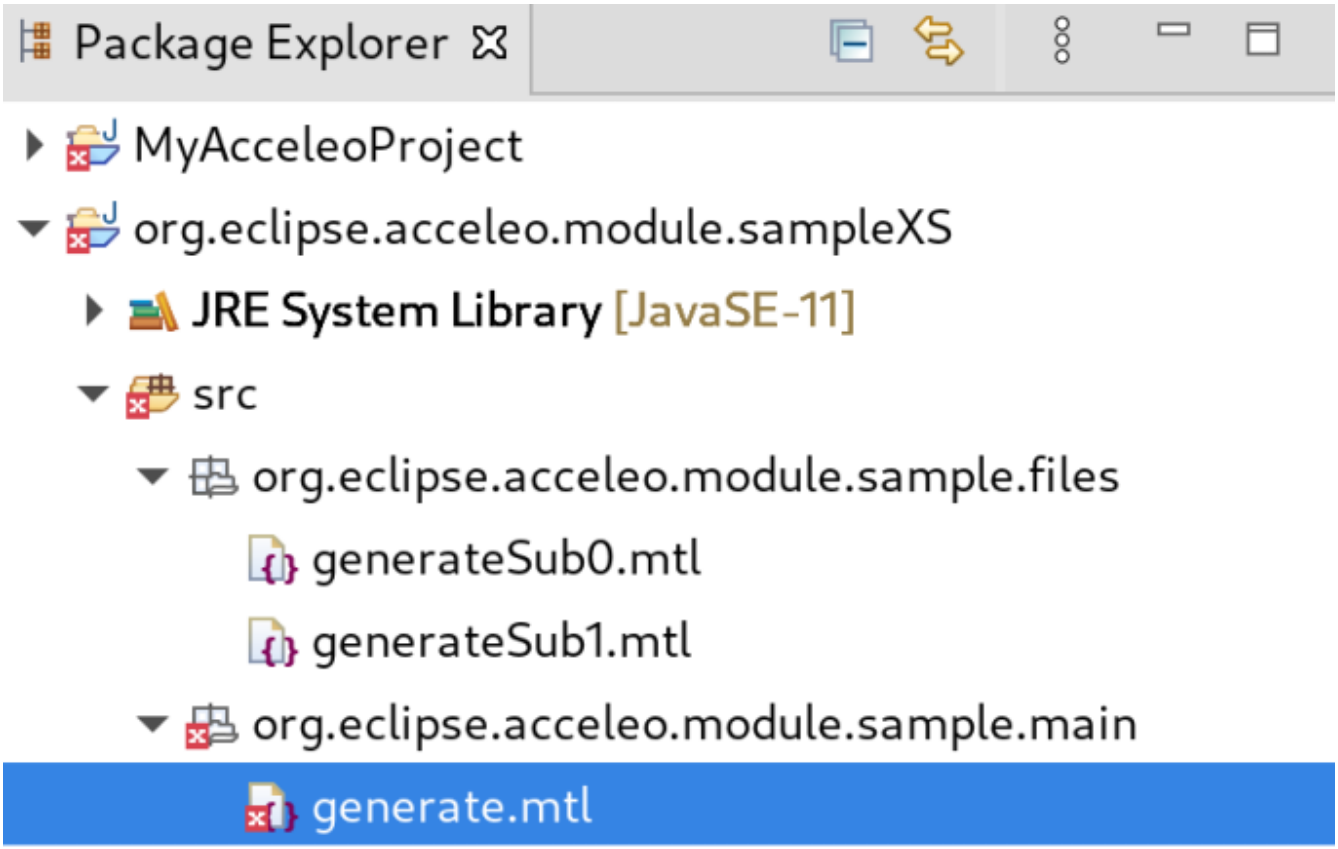
4.1.3. Real Time Errors Detection

Obviously, Acceleo displays error markers when errors are detected. Acceleo displays error markers whenever a module file is not valid, whatever the reason. Errors appear in the **Problems** view (generally displayed at the bottom of the perspective), and double-clicking on an error in this

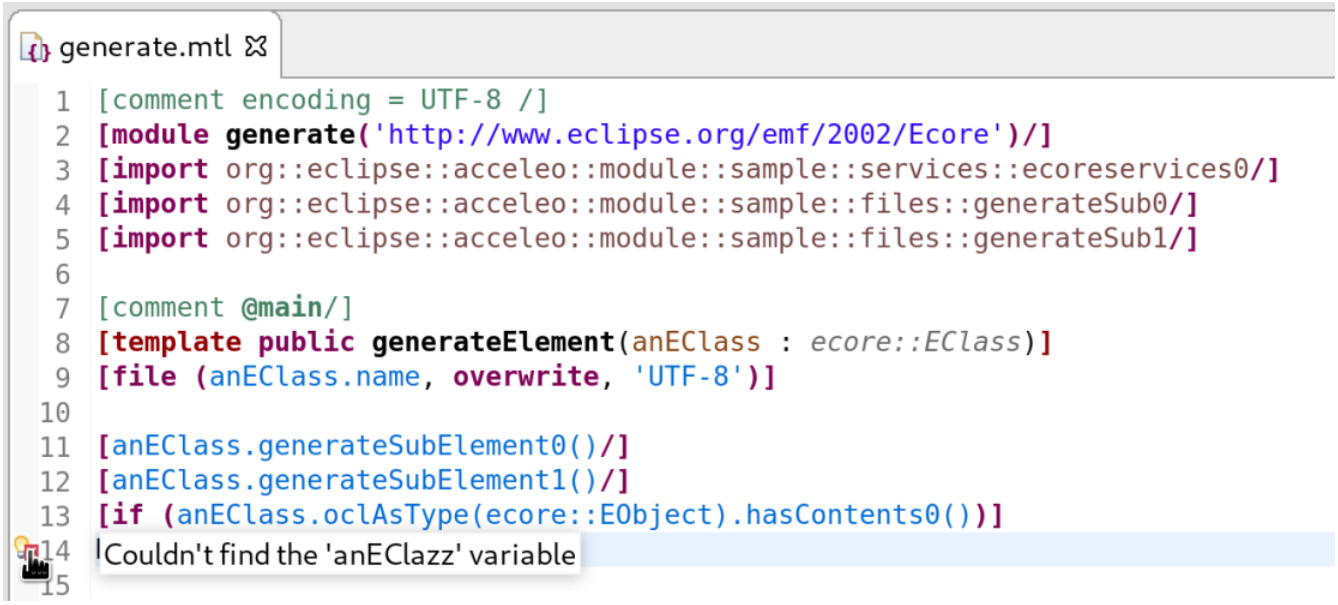
view directly takes you to the file where it is located.



Files with errors also appear with an error decorator.



Just hover the marker in the editor margin with the mouse to get a tooltip to appear with an explanation of the problem.



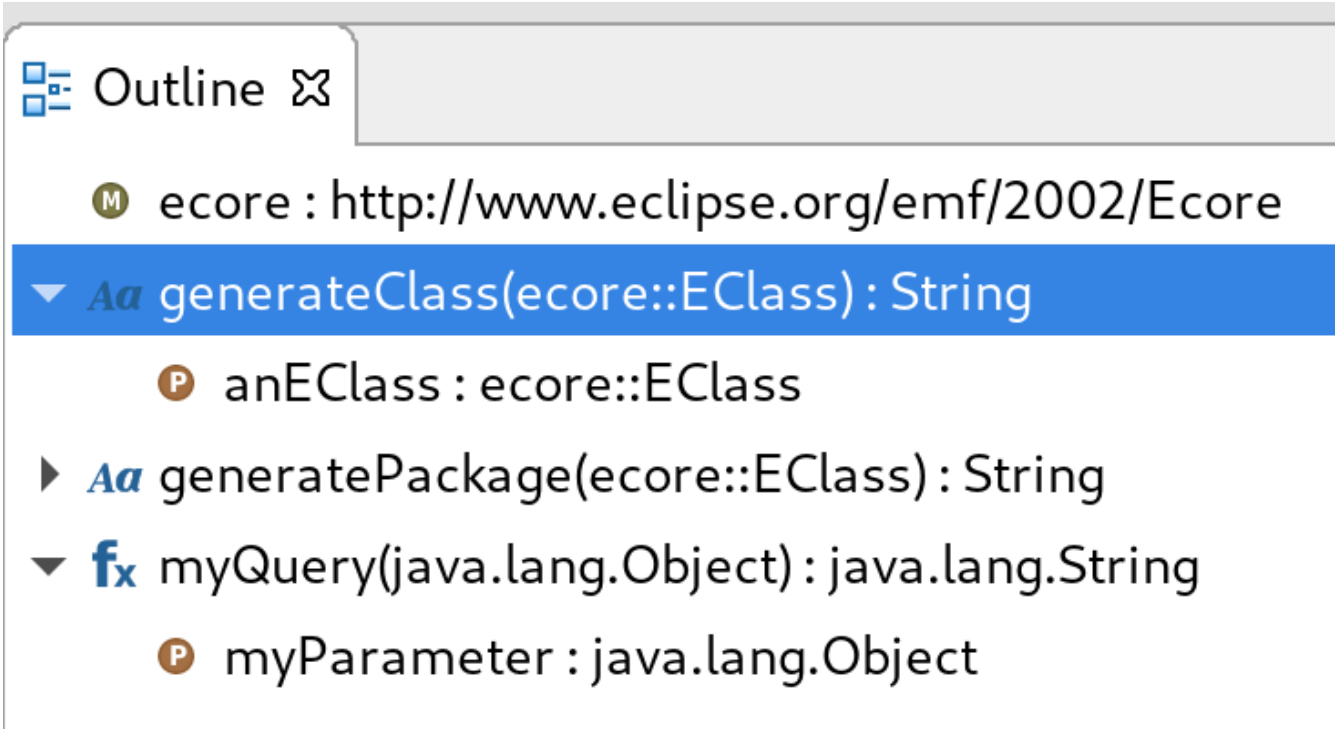
4.1.4. Outlines

One of the great benefits of modern IDE tooling is the capacity to easily navigate in code from elements to their declarations and, vice-versa, from declarations to usages.

The dynamic outline

The traditional Eclipse **Outline** view is used by Acceleo to display the module's structure metamodels, templates, queries can be seen there, and double-clicking on any of them places the

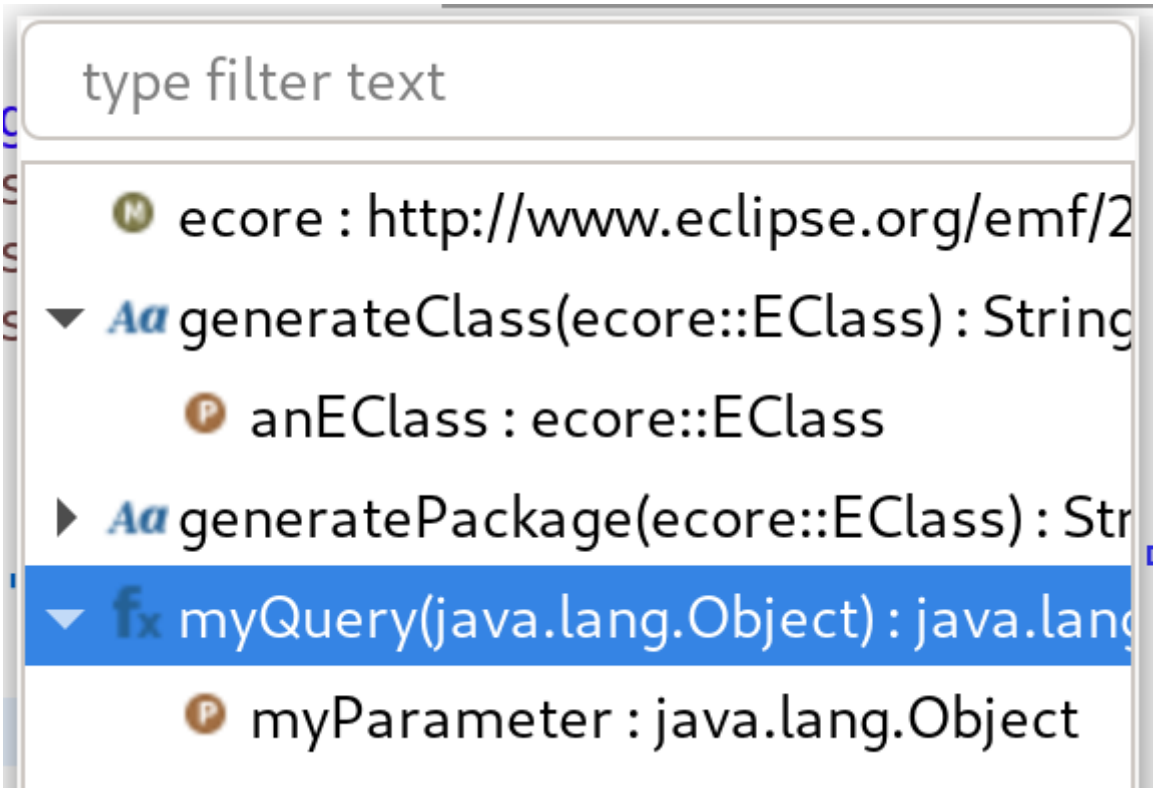
cursor at the corresponding position in the module (in the editor).



The quick outline

The quick outline, which can be displayed using `Ctrl + O`, displays just the necessary information to access any element in the current module. So, hitting `Ctrl + O` displays a pop-up with a list of templates and queries.

A text field at the top allows you to quickly filter the content in order to easily find what you are looking for.



4.1.5. Open Declaration

The traditional shortcut `F3` is supported by Acceleo, along with `Ctrl + click`, which both take you to the declaration of the selected or clicked element. This is supported for all kinds of elements: templates, queries, metamodels, metamodel elements, EOperations, etc.

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.eclipse.org/emf/2002/Ecore')/]
3  [import org::eclipse::acceleo::module::sample::services::ecoreservices0/]
4  [import org::eclipse::acceleo::module::sample::files::generateSub0/]
5  [import org::eclipse::acceleo::module::sample::files::generateSub1/]
6
7  [comment @main/]
8  [template public generateClass(anEClass : ecore::EClass)]
9  [file (anEClass.name, overwrite, 'UTF-8')]
10
11 [anEClass.generateSubElement0()/]
12 [anEClass.generateSubElement1()/]
13 [if (anEClass.oclAsType(ecore::EObject).hasContents0())]
14 Name: [anEClass.name/]
```

4.1.6. White spaces

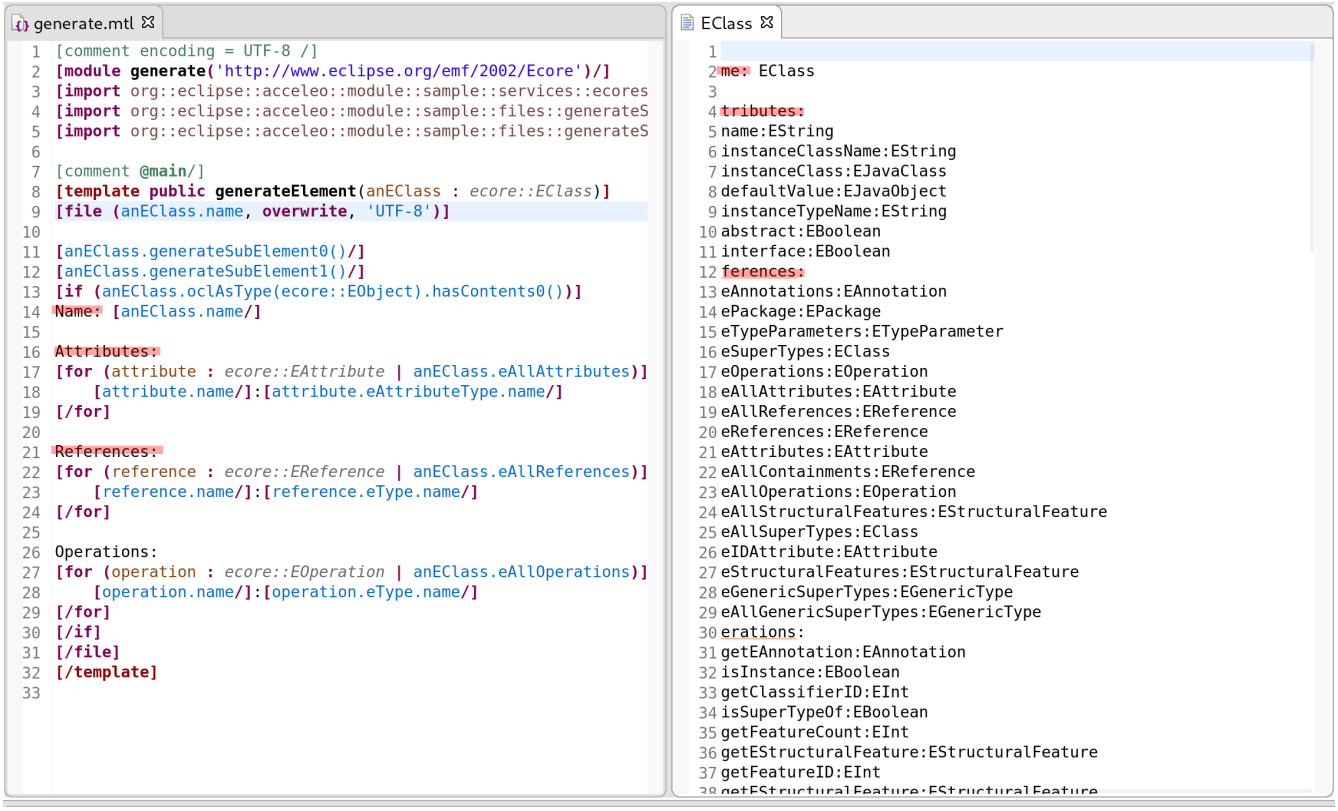
When generating text, and especially code, white spaces and indentation is an important point. In order to keep template code indentation from interfering with the generated output, a few rules applies:

- each block has a **mandatory indentation** of **two characters** than will not be generated in the output (in yellow below)
- when generating a block if the last generated line is not empty, it is repeated at the beginning of each line generated by the block (in red below)

```
[module test('http://www.eclipse.org/emf/2002/Ecore')/]¶
¶
[import test::ModuleServices/]¶
[import test::otherModule/]¶
¶
[comment @main /]¶
[template public myTemplate(myParam : ecore::EPackage)]¶
  [file (myParam.name + '.txt', overwrite)]¶
    - [for (myVariable1 : ecore::EClassifier | myParam.eClassifiers)]¶
        some text [myVariable1.name/]¶
    [/for]¶
  [/file]¶
[/template]¶
```

```
  some text World¶
  some text MultiNamedElement¶
  some text NamedElement¶
  some text Producer¶
  some text Adress¶
  some text Company¶
  some text ProductionCompany¶
  some text Restaurant¶
```

In a template you have to **let 2 whitespaces at the beginning of each line**, else the generated content will be truncated by 2 characters.



4.2. The Acceleo Debugger

4.2.1. Breakpoints

To add a breakpoint somewhere in a template, just double-click in the left margin on the line where you want to add the breakpoint. A nice bluish marker should appear, which should be very familiar to eclipse users.

generate.mtl

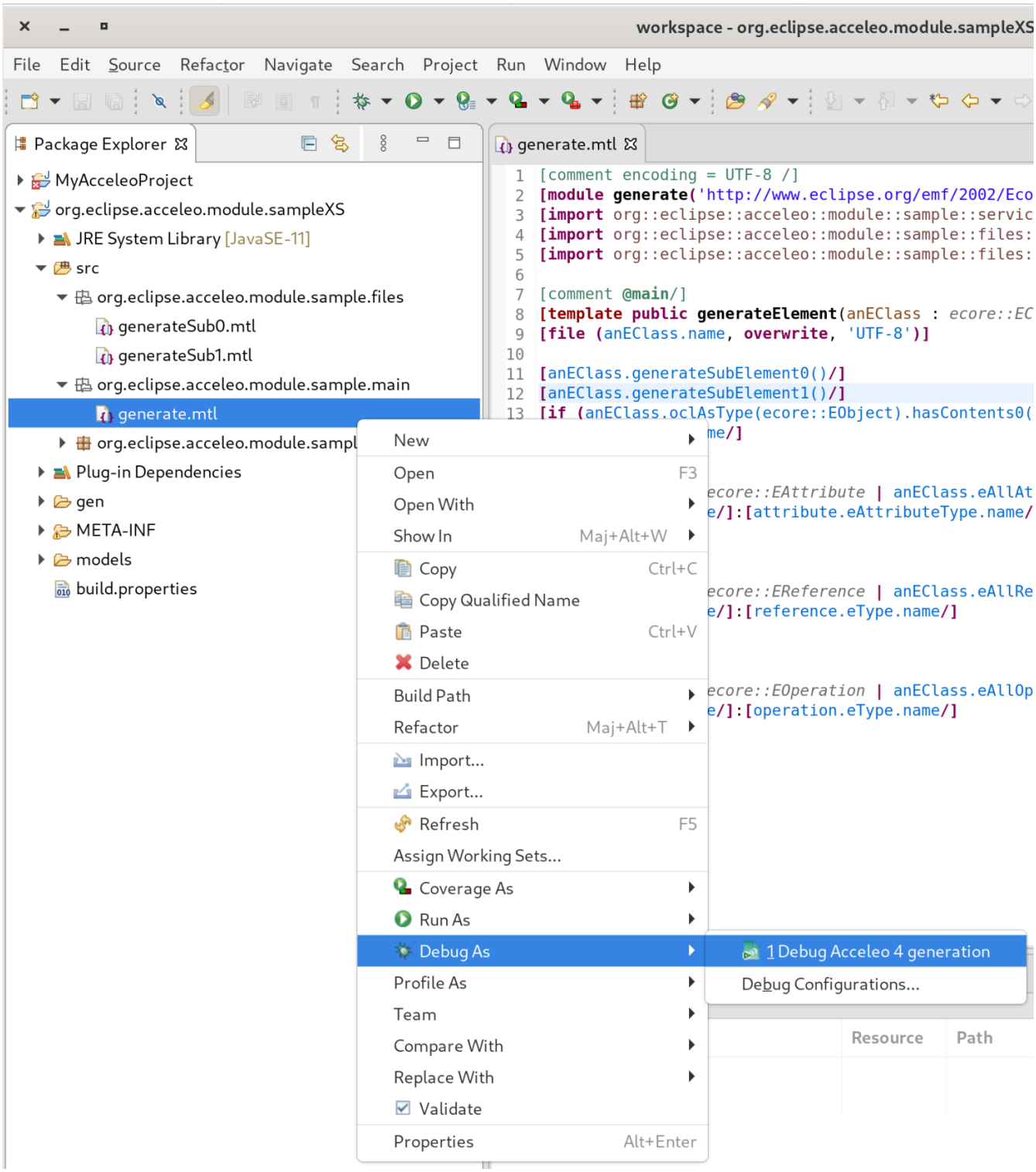
```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.eclipse.org/emf/2002/Ecore')/]
3  [import org::eclipse::acceleo::module::sample::services::ecoreservices0/
4  [import org::eclipse::acceleo::module::sample::files::generateSub0/]
5  [import org::eclipse::acceleo::module::sample::files::generateSub1/]
6
7  [comment @main/]
8  [template public generateElement(anEClass : ecore::EClass)]
9  [file (anEClass.name, overwrite, 'UTF-8')]
10
11 [anEClass.generateSubElement0()/]
12 [anEClass.generateSubElement1()/]
13 [if (anEClass.oclAsType(ecore::EObject).hasContents0())]
14 Name: [anEClass.name/]
15
16 Attributes:
17 [for (attribute : ecore::EAttribute | anEClass.eAllAttributes)]
18     [attribute.name/]:[attribute.eAttributeType.name/]
19 [/for]
20
21 References:
22 [for (reference : ecore::EReference | anEClass.eAllReferences)]
23     [reference.name/]:[reference.eType.name/]
24 [/for]
25
26 Operations:
27 [for (operation : ecore::EOperation | anEClass.eAllOperations)]
28     [operation.name/]:[operation.eType.name/]
29 [/for]
30 [/if]
31 [/file]
32 [/template]
```

Conditional breakpoints are not supported.

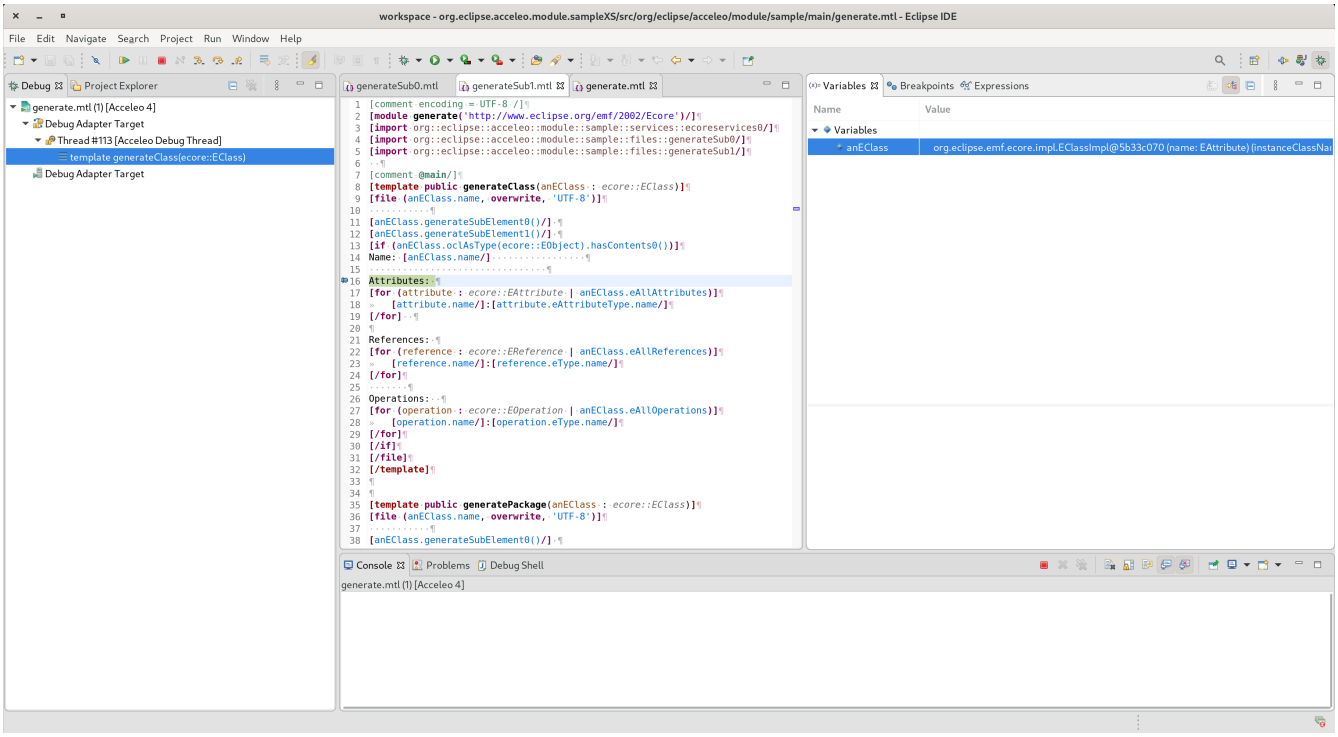
4.2.2. Step by step execution

To debug an Acceleo generation:

- Right-click on your `.mtl` file, and select **Debug As** > **Debug Acceleo 4 generation**.

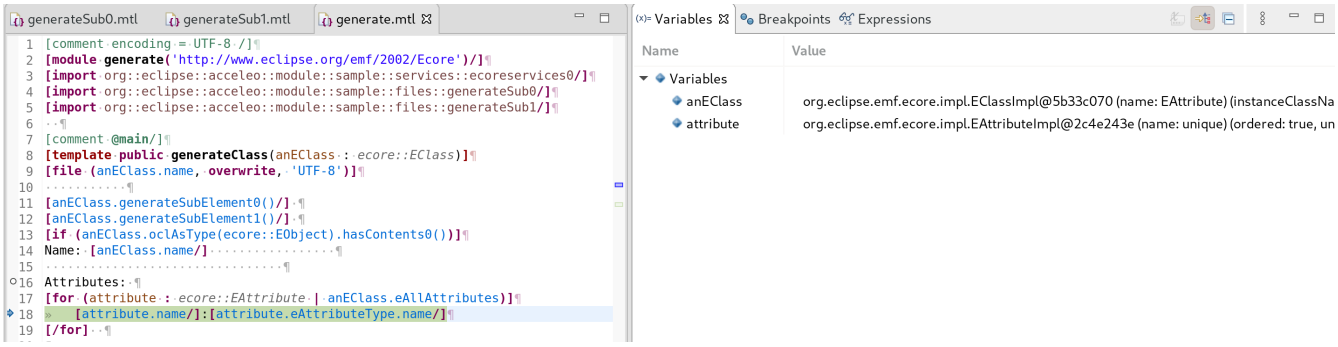


It is actually recommended to switch to the **Debug Perspective**, which is really more appropriate to debug executions.



The debug perspective should be quite familiar to people used to the eclipse IDE. The **Debug** view (on the top left) displays the stack of the current execution. Clicking on any element of the stack will simultaneously display the corresponding Acceleo code in the edition area.

The **Variables** view displays currently accessible variables. In the example below, the execution has met a breakpoint when computing `Attributes` for a class, so the current input is a class. The Variables view tells us that the current class is called `EAttribute`.



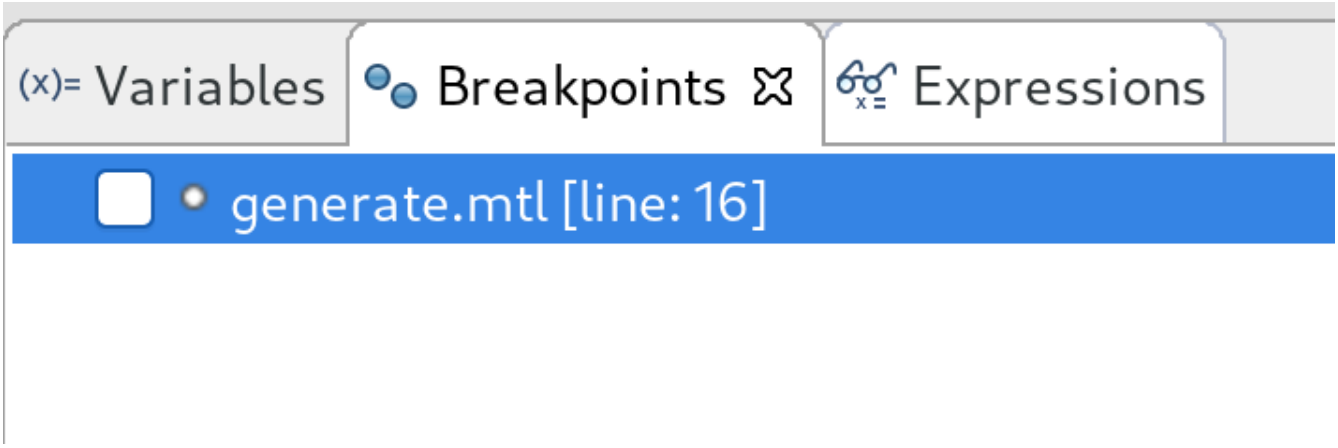
The debugger cannot step into AQL expressions only Acceleo elements are visible in the Variables view.

As usual, it is possible to:

- **Step into** a template (`F5`);
- **Step over** a template (`F6`);
- **Step Return** (`F7`), which means "go to the end of the current template";
- **Resume** execution (`F8`);
- **Stop** execution (`Ctrl + F2`). The icons above the **Debug** view serve the same purpose.

Acceleo breakpoints can be temporarily deactivated, thanks to the **Breakpoints** view. Just uncheck

the checkbox in front of a breakpoint to deactivate it. Here is an example of a deactivated breakpoint in this view:



5. The Acceleo Language

You can read to the language documentation to understand Acceleo 4 language specificity. If you are looking for the syntax quick reference you can check the syntax page.

5.1. Intended Audience

This guide was written to describe the elements composing the Acceleo 4 templating language. This will not delve into the Acceleo Query Language (AQL) description.

5.2. Module

An Acceleo module is defined in its own file and is the main container for generation templates (that will generate text) and queries (which purpose is to extract information from the input models).

The name of the module will be qualified according to its location in the source folder of the project.

An Acceleo file must start with the module declaration in the following format:

```
[**
<module documentation>
@author <author name>
@version <version number>
@since <first version in which this module appeared>
/]
[module <module name>('http://metamodel/URI/1', 'http://metamodel/URI/1',
<additional_URIs>) extends <other module qualified name>]
```

Module documentation

The documentation of a module is optional, and can contain both the description of the module

and optional metadata such as the author and version number.

Module name

The module name can only contain alphanumeric (and underscore _) characters and cannot start with a number.

Metamodel URIs

The metamodel(s) from which this module will take types. The list has to be exhaustive: if multiple connected metamodels are used, then all must be listed in the module declaration.

Extends

The qualified name of another module. Acceleo allows you to override *protected* and *public* visibility templates from the extended module. Extending multiple modules is not allowed. **Note** that the extended module's name has to be qualified, even when overriding modules located in the same package.

See also the Module syntax documentation.

5.2.1. Imports

An Acceleo module can import any number of other modules or Java service class. All *public* visibility templates and queries from the imported module(s) can be called by the importer, for Java class all public methods can be used a service.

Importing modules can be done right after the module declaration line and requires the following format:

```
[import <other module or Java class qualified name>/]
```

Note that modules can only be referenced through their qualified name, even if they are located in the same package.

See also the Import as well as the Module Reference syntax documentation.

5.2.2. Module Elements

Following the imports declaration, any number of module element can now be written as the body of our Module. Please refer to Template and Query sections below for more information.

5.3. Template

A Template is a set of Acceleo statements used to generate text. It is delimited by `[template ... ][/template]` tags.

The template signature must include the visbility and the name, and can optionally define a post-treatment expression.

```
[**
<template documentation>
@param class <documentation of the parameter>
/]
[template public generate(class : ecore::EClass) post (self.trim())]
[/template]
```

visibility

One of

- public : Public templates will be visible from all modules importing or extending the module declaring them. They can be overridden by extending modules.
- protected : Protected templates will only be visible from extending modules and can be overridden.
- private : Private templates are only visible by the defining module and cannot be overridden.

name

The name of the template. Only alphanumeric (and underscore `_`) characters are allowed, and the name cannot start with a number.

arguments

Arguments follow the format for variables.

post

The post-treatment expression will be called on the result of the template (stored in variable `self`) and needs to be applicable to a String.

The result of a template is always a String.

See also the Template syntax documentation.

5.3.1. Main template

Acceleo defines a special metadata tag on templates to specify the main entry point(s) of a generation, the template(s) that will be called first during the workflow. This metadata tag, `@main`, needs to be in the comments preceding the template, or within the template documentation

Such templates do not necessarily contain a File block themselves.

Example:

```
[**
<template documentation>
@param class <param documentation>
@main
/]
[template public generate(class : ecore::EClass)]
[file (class.name + '.txt', overwrite)]
Class [class.name/] structural features:
[for (feature | class.eStructuralFeatures)]
[feature.name/]
[/for]
[/file]
[/template]
```

5.3.2. File Block

File blocks are used to tell the Acceleo engine it must generate the body of the `[file ⋯]` block in an actual file.

```
[file (<uri>, <open mode>, <charset>)][/file]
```

uri

An AQL expression denoting the output file name. Must evaluate to a String.

open mode

The open mode for the file. This can be one of:

- **(Not Supported Yet)** append : Append to the end of the file if it already exists, create it otherwise.
- overwrite : Overwrite the whole file if it already exists, create it otherwise.
- create : Do not change the file if it already exists, create it otherwise.

charset

This can be used to tell Acceleo which charset to use for the generated file. This is optional and will default to UTF-8

Example:

```
[template public generate(class : ecore::EClass)]
[file (class.name + '.java', append, 'UTF-8')]
[/file]
[/template]
```

Note that the file will only be generated if the engine actually evaluates the file block. For example, if the template containing that block is never called, or if the file block is included in an `If` block

which condition evalutes to `false`, then no file will be created.

See also the File Statement syntax documentation.

5.3.3. For loops

For loops in Acceleo need to be expressed using the following syntax:

```
[for (<variable> | <iteration expression>) separator(<separator expression>)]...[/for]
```

variable

The variable follows the format for variables. The variable type must match the result of the iteration expression. If the expression returns a collection of Strings, then the variable will be of type String. Because of this, typing the variable is optional.

iteration expression

An AQL expression returning a collection of elements on which to iterate. If the expression doesn't return a collection but a single element, it will be used to iterate only once.

separator

The separator expression will be evaluated to insert content in-between the content generated for each iteration of the for body. It will not be generated if the for loop doesn't generate text or only generates one iteration worth of content.

Example:

```
[template public generate(class : ecore::EClass)]
[for (feature | class.eStructuralFeatures)]
[/for]
[/template]
```

See also the For Statement syntax documentation.

5.3.4. If conditions

The `If` statement in Acceleo uses the following syntax:

```
[if (condition)]
[elseif (condition)]
[else]
[/if]
```

`elseif` and `else` are both optional. If one of the `condition` expressions does not evaluate to a boolean an error will be logged and nothing will be generated for that `if` block.

See also the If Statement syntax documentation.

5.3.5. Let block

Acceleo `Let` blocks use the following syntax:

```
[let <variable1> = <init expression>, <variable2> = <init expression>]
[/let]
```

`Let` blocks allow template writers to define temporary variables that will be visible within the scope of the block.

The variables follow the format for variables. All of their typing is optional since they must match the return type of their initialization expression.

Note that all variables are immutable. Nested `Let` blocks can override the value of a variable within their own scope, but the variable will go back to its former value once outside of the nested `Let`. This will produce a warning during validation.

See also the Let Statement syntax documentation.

5.3.6. Protected Area

A protected area defines a set of statements that should only be generated if the file doesn't exist on disk or it does not contain an area with the specified protected area's identifier.

Protected areas allow module writers to create a "safe" part of the generated file that can be modified directly in the generated file, without fear of these manual modifications to be lost during subsequent generations.

```
[comment @main/]
[template public generate(class : ecore::EClass)]
[file (class.name + '.java', overwrite, 'UTF-8')/]
// [protected (class.name + ' imports')]
imports java.util.List;
// [/protected]

public class [class.name.toUpperFirst()/] {

}
[/file]
[/template]
```

The expression within the protected block's signature serves as the protected area's identifier and **must be unique** in the generated file's scope.

There must be nothing present on the line after the protected area's signature. Otherwise, everything following said signature will be considered to be part of the area's identifier by the engine and the code will not be properly protected.

Please also note that the protected area's first and last line are marked as comments in the generated code (by generating `//` at the start of their respective line, since this is Java code). This is to avoid generating invalid Java code as the markers will be present in the generated file's contents.

5.3.7. Variable

Acceleo 4 variables use the AQL syntax and inference logic for their typing.

```
<name> : <type>
```

name

Name of the variable. Only alphanumeric (and underscore`_`) characters are allowed, and the name cannot start with a number.

type

Type of the variable. Four different kind of types are accepted

- primitive : Integer, Double, String, Boolean
- collection : Sequence, OrderedSet. Collection types have to be further specified with their content types, such as Sequence(String) for a list of String elements.
- eclassifier : in the form <epackage_name>::<classifier_name> such as ecore::EClass.
- union type : in the form {<epackage_name>::<classifier_name> | <epackage_name>::<classifier_name> | ...}. This kind of typing describes a variable that can be either one of the *n* specified classifiers. e.g. {ecore::EAttribute | ecore::EReference }.

5.4. Query

A query is a re-useable AQL expression that can return any type of Object. They are commonly used to extract information from the input models. A query is enclosed in a `[query ⋯]` tag.

The query signature must include the visibility and its name.

```
[**
<query documentation>
@param class <documentation of the parameter>
/]
[query public getPublicProperties(class : uml::Class) : Set(uml::Property) =
    class.attribute->select(property : uml::Property | property.visibility =
uml::VisbilityKind::public)
/]
```

visibility

One of

- public : Public templates will be visible from all modules importing or extending the module

declaring them. They can be overridden by extending modules.

- protected : Protected templates will only be visible from extending modules and can be overridden.
- private : Private templates are only visible by the defining module and cannot be overridden.

name

The name of the query. Only alphanumeric (and underscore _) characters are allowed, and the name cannot start with a number.

arguments

Arguments follow the format for variables.

return type

The return type describes the kind of object this query is expected to return. If the expression does not return an object of the accurate type, the evaluation will fail at runtime.

See also the Query syntax documentation. = Acceleo Acceleo 4 Syntax

6. Preface

This document describe the syntax of an Acceleo 4 module.

The syntax is described using the Backus Naur from (BNF).

7. Syntax

7.1. Comment

A comment can be used to document any part of the Module. It generates nothing if placed directly or indirectly in a File Statement. For simplification comments will not be present in the BNF representation of the grammar.

```
Comment = '[comment ' ... '/]'
```

7.2. Module

The module is the top level element of a .mtl file. It represent a namespace declaring Template and Query. The name of the module is qualified by the location of the file in the source folder.

```
Module =

(Module Documentation)* '[module ' Identifier '(' Metamodel ',' (Metamodel)* ')'
('extends ' Module Reference)? '/]'

Import*

Module Element*
```

7.3. Identifier

An identifier is used to name elements that need to be identified, or reference element that can be identified.

```
Identifier = [a-zA-Z_][a-zA-Z_0-9]*
```

7.4. Module Documentation

The module documentation should contains a description of the Module.

It can also contain metadata such as the author, version, and since (the version since this Module exists).

```
Module Documentation =

'[**'

...

('@author' ...)?

('@version' ...)?

('@since' ...)?

...

'/]'
```

7.5. Metamodel

This is the declaration of metamodels used by the module. Metamodels are referenced using their EPackage nsURI between simple quote.

```
Metamodel = '\'' ... '\''
```

7.6. Import

This allows a module to import other Module or service classes.

```
Import = '[import ' Module Reference '/]'
```

7.7. Module Reference

The module reference is a qualified reference to a Module

```
Module Reference = Module Qualified Name
```

7.8. Module Qualified Name

A module's name is qualified according to its location in the source folder of a project.

```
Module Qualified Name = Identifier ('::' Identifier)*
```

7.9. Module Element

A module element is either a Template or a Query.

```
Module Element = Template | Query
```

7.10. Template

A template returns a String produced using its contained Statement, it can be called as a service. It can be preceded by a Module Element documentation.

Also a Module can contain a template used as entry point of the generation. This template will be identified with a Comment preceding the template and containing the tag '@main'.

```
Template =

'[template ' Visibility Identifier '(' Parameter(',' Parameter)* ')' ('?' AQL
Expression)? ('post (' AQL Expression ')')? ']'

(Statement)*

'[/template]'
```

7.11. Visibility

The visibility defines the scope in which a Module Element can be called as a service.

```
Visibility = 'private' | 'protected' | 'public'
```

7.12. Parameter

A parameter is used to pass a value from the caller to a callee. This value can be later referenced using its identifier.

```
Parameter = Identifier ':' AQL Type Literal
```

7.13. Statement

A statement is a directive used to produce an output or control the execution flow.

```
Statement =

File Statement | For Statement | If Statement | Let Statement | Protected Area |
Expression Statement | Text Statement
```

7.13.1. File Statement

This statement is used to start the generation of a new file. Strings returned by a statement contained directly or indirectly in the execution flow, will be generated into that file.

The file statement itself returns an empty String.

```
File Statement =

'[file ' '(' AQL Expression ',' [Open Mode Kind] (',' AQL Expression)? ')' ']'

(Statement)*

'[/file]'
```

7.13.2. For Statement

This statement loops over a list of values and return the concatenation of all returned String.

```
For Statement =

'[for ' '(' Identifier (':' AQL Type Literal)? '|' AQL Expression ')' ['separator('
AQL Expression ')'] ']'

(Statement)*

'[/for]'
```

7.13.3. If Statement

This statement create a branch in the execution flow and return the String of one of its branch according to the AQL Expression evaluated to true. If a condition doesn't evaluate to a boolean an empty String is generated and an error is logged.

```
If Statement =

'[if ' '(' AQL Expression ')' ']'

(Statement)*

('[elseif ' '(' AQL Expression ')' ']'

(Statement)*)*

('[else]'

(Statement)*)?

'[/if]'
```

7.13.4. Let Statement

This statement allows to compute one or more AQL Expression and reference their value using an

identifier. It can be used to improve readability of the template or increase performance when using the same AQL Expression many times in a block of Statement.

```
Let Statement =

'[let ' Identifier (':' AQL Type Literal)? '=' AQL Expression (',' Identifier (':' AQL
Type Literal)? '=' AQL Expression)* ']'

(Statement)*)?

'[/let]'
```

7.13.5. Protected Area

This statement declares an identified area in the generated file. If the generated file exists and a protected area with the same identifier exists in its contents, then the existing content of this area is directly returned. If it doesn't exist, then the concatenation of the body's statements results is returned.

```
Protected Area =

'[protected ' '(' AQL Expression ')' ']'

(Statement)*)?

'[/protected]'
```

7.13.6. Expression Statement

This statement returns the String representation of the evaluation of its AQL Expression.

```
Expression Statement = '[' AQL Expression '/]'
```

7.13.7. Text Statement

This is any other text outside of '[' and ']'.

7.14. AQL Expression

This is an Acceleo Query Language expression. It is used to navigate through models and call services. In the context of Acceleo, Template and Query can be called as services.

See the AQL documentation for more details about the language itself, the full list of standard services, and the differences with Acceleo/MTL.

7.15. AQL Type Literal

This is a type literal as defined in the Acceleo Query Language.

7.16. Query

A query references an AQL Expression with parameters and can be called as a service. It can be preceded by a Module Element documentation.

```
Query =

'[query ' Visibility Identifier '(' Parameter(',' Parameter)* ')' ':' AQL Type Literal
'=' AQL Expression '/]'
```

7.17. Module Element documentation

The documentation of a Template or a Query.

```
Module Element documentation =

'[**'

...

'@param ' ...

...

'/]'
```

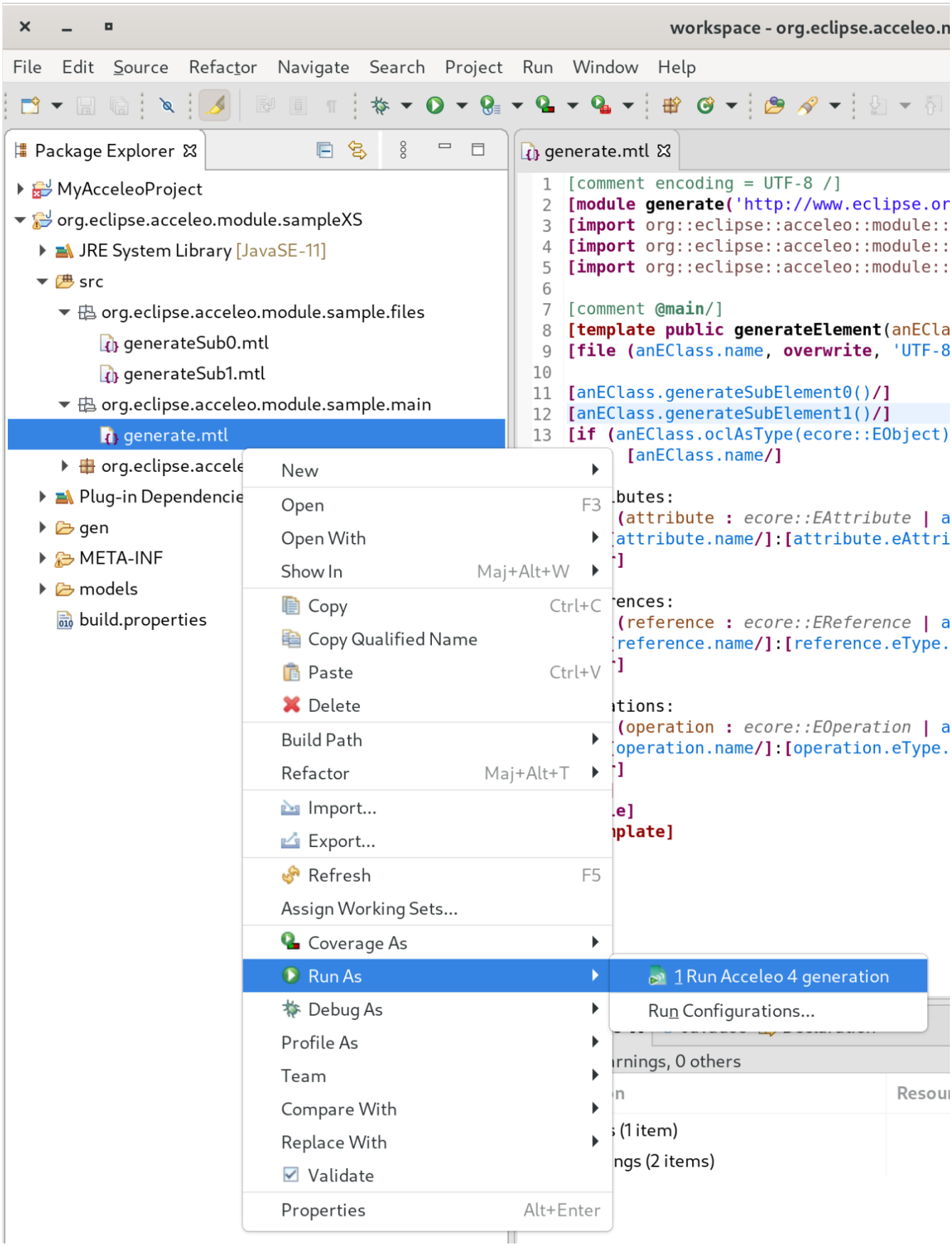
8. Launching an Acceleo generation

This section describe how to launch a generation from [eclipse] or Maven/Tycho. Note that you can also use the [_debugger] to run a module or directly use Acceleo 4 programmatically, see Generation.

8.1. Eclipse

IMPORTANT	*Before you Start* Use a **Java Project** and the classical **Java Perspective** for writing your Acceleo templates `.mtl` in the **src** folder.

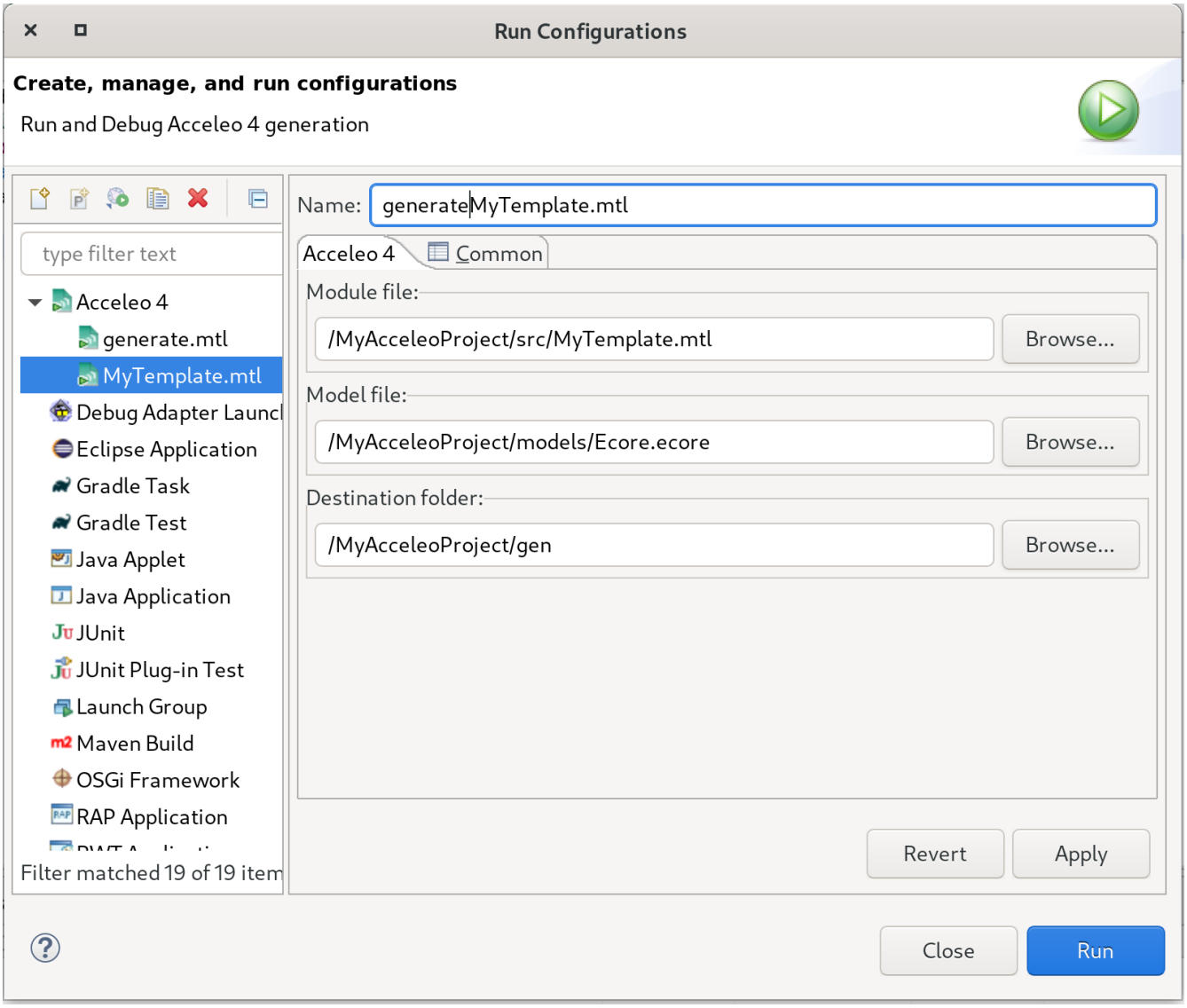
In order to launch an Acceleo generator, you just have to right-click on your main module and use the **Run As** menu.



From there the **Launch Configuration** menu will open. If you already have created a launch configuration you can access the launch configuration menu from the **Run** > **Run Configurations...** menu. In this menu, you will have access to the Acceleo Launch configuration. You just have to select:

- your main **module file**,

- your input **model file**,
- the **destination folder** of the generation.



INFO: Check that your generation is finished by opening the **Console** view. You should see a message like this `<terminated> generate.mtl[Acceleo4]`.

8.2. Maven/Tycho

Acceleo provides a specific eclipse application that can be used from command line or tycho in order to start a generation. In order to use the launcher, you will need to first package all of your generator projects into an update site, then use this update site as input of the application.

An example is available in the Acceleo repository and can be checked out at: https://git.eclipse.org/c/acceleo/org.eclipse.acceleo.git/tree/examples/MavenLauncher

This example includes:

- A generator project `org.eclipse.acceleo.aql.launcher.sample`,
- A feature including this project `org.eclipse.acceleo.aql.launcher.sample-feature`,
- An update site including this feature `org.eclipse.acceleo.aql.launcher.sample-site`,

- A `pom.xml` file that can be used to both package the generator and start the generation.

The packaging can be started through the command line `mvn clean verify`. Once packaged, generation can be started through the command line `mvn clean verify -Pgenerate`.

The application takes a number of arguments that will have to be customized through the pom.xml file:

```
<applicationsArgs>
    <args>-application</args>
    <args>org.eclipse.acceleo.aql.launcher.AcceleoLauncher</args>
    <args>-data</args>
    <args>${project.build.directory}/wks</args>
    <args>-bundle</args>
    <args>org.eclipse.acceleo.aql.launcher.sample</args>
    <args>-module</args>
    <args>org::eclipse::acceleo::aql::launcher::sample::main</args>
    <args>-models</args>

<args>platform:/plugin/org.eclipse.acceleo.aql.launcher.sample/model/sample.xmi</args>
    <args>-target</args>
    <args>file:/${project.build.directory}/generated/</args>
</applicationsArgs>
```

application

Standard Eclipse argument, this tells Eclipse which application it needs to run. The specific launcher for Acceleo generations is `org.eclipse.acceleo.aql.launcher.AcceleoLauncher`.

data

Standard Eclipse argument, this can be used to modify the workspace path.

bundle

This needs to be the identifier of the bundle containing the main module for this generation.

module

The starting point of the generation that is to be started. This needs to the qualified name of the module containing an "@main"-annotated template.

models

The URI of the models that will be fed to the main module of the generation. This cannot be empty and needs to be an URI that eclipse is capable of resolving. In this example we're using a `platform:/plugin/⋯` URI since we've bundled the input model into our generator project.

target

The destination URI for this generation. Generated files will use this folder as their root to resolve against. **Note** that this needs to end in a trailing `/`.

9. Using Acceleo 4 programmatically

Acceleo 4 can be used programmatically and for instance integrated in other products.

9.1. Parsing

```
URI destination = URI.createURI(...);
environment = new AcceleoEnvironment(new DefaultGenerationStrategy(), destination);
IQualifiedNameResolver moduleResolver = new
ClassLoaderQualifiedNameResolver(getClass().getClassLoader(),
environment.getQueryEnvironment());
environment.setModuleResolver(moduleResolver);
Module module = environment.getModule(qualifiedName);
```

9.2. Validation

```
AcceleoValidator validator = new AcceleoValidator(environment);
List<IValidationMessage> messages = validator.validate(astResult,
qualifiedName).getValidationMessages();
```

9.3. Completion

```
AcceleoCompletor completor = new AcceleoCompletor();
String source = ...;
List<ICompletionProposal> proposals = completor.getProposals(environment, source,
position);
```

9.4. Generation

```
AcceleoEvaluator evaluator = new AcceleoEvaluator(environment);
ResourceSetImpl rs = new ResourceSetImpl();
Resource model = rs.getResource(..., true);
AcceleoUtil.generate(evaluator, environment, module, model);
```

9.5. Unit test module

You can unit test your modules using the same JUnit test suite we are using for the development of Acceleo 4. You will simply need to create a class extending the class org.eclipse.acceleo.tests.utils.AbstractEvaluationTestSuite and create a folder with you test folders. Those folders need to respect a naming convention. You can find a working example with the class FileStatementTests and the corresponding folder.

10. Migrate Acceleo 3 templates to Acceleo 4

This section was written to describe the behavior changes between the two versions of Acceleo, for the purpose of migrating from one to the next.

10.1. How to migrate Acceleo 3 templates to Acceleo 4

10.1.1. The migration tool

The migration tool consists on a java standalone utility deployed as a jar file, "migrator.jar". There is a library folder aside containing all of the required jars (Acceleo 3, Acceleo 4) required to perform a migration.

10.1.2. Launching a migration

The migration tool requires a fully built Acceleo 3 project as input: the project folder must contain a bin/ folder with all of the compiled (.emtl) versions of the .mtl source files.

The migration jar can be run in command line with the following arguments. We assume we are in the folder containing the migration jar, and that the Acceleo project is at the same level. There is also an empty "output" folder to receive the result of the migration:

```
java -jar migrator.jar <myproject>/<sourcefolder> <output_folder>
```

For instance, for an Acceleo 3 project in which the source templates are in the src/ folder (anywhere in the sub-folders of src/):

```
java -jar migrator.jar myProject/src output
```

The converted .mtl files will be created in the output folder, accordingly to the folder structure in the source project.

10.2. Language Changes

10.2.1. Modules

module name

Acceleo 3 allowed users to set the fully qualified name of the module in its declaration, such as

```
[module qualified::module::name(...)]
```

Acceleo 4 forbids anything other than the actual module name and this will thus become.

```
[module name(...)]
```

Both versions require the module name to be equal to the name of the containing file minus the extension.

multiple inheritance

The Acceleo 3 module declaration allowed users to declare multiple "extends" modules:

```
[module qualified::module::name(...) extends another::module, and::a:second::module]
```

Acceleo 4 modules can only define a single extended module.

Only the first extended module was taken into account in Acceleo 3 so the migration just strips all but the first extended modules' name.

module imports

Acceleo 3 allowed imports of modules through unqualified names. This is forbidden in Acceleo 4 and the fully qualified names of the imports are required.

10.2.2. Templates

Important notes: Every template which had duplicated signatures (same name, same argument list, different pre-condition) needs to be manually checked after migration.

- Acceleo 4 does not allow duplicated signatures and has no pre-condition, so duplicated templates might have to be combined into a single one with their pre-condition as an `If-Else` block inside.
- Initialization blocks could be different on duplicated templates, depending on the pre-condition. The migrated `Let` block thus needs to be different per branch of the above `If-Else`.

Acceleo 4 templates will automatically override their super-module templates if the signature matches. This was not the case in Acceleo 3 and matching templates thus need verified manually.

Overriding

Acceleo 3 templates allowed users to override an arbitrary template through the "overrides" keyword:

```
[template public aTemplateName(...) overrides anotherTemplateName]
```

Acceleo 4 only allows a template of name "xyz" to override a template from the extended module with the same "xyz" name.

Furthermore, Acceleo 4 templates will automatically override their super-module's public or protected templates if they have the same signature.

Pre-conditions

Acceleo 4 does not allow pre-conditions on templates.

Post-treatment

Acceleo 3 used an implicit String-typed variable so the post expression could be a simple call without a variable:

```
[template public aTemplateName(...) post (trim())]
```

Acceleo 4 doesn't allow implicit variables. The result of the template call will be stored in the `self` variable and the migration will thus transform this expression into:

```
[template public aTemplateName(...) post (self.trim())]
```

Init block

Acceleo 3 allowed variable initialization blocks on the template:

```
[template public aTemplateName(...) {var1 = 'string'; var2 = param1.feature;}]
```

Acceleo 4 removes the initialization blocks altogether, so the variable declarations must be manually transformed into a `let` block at the start of the template (this is not supported by the migration tool).

Namesakes

Acceleo 3 allowed multiple templates with the same name and same argument list to co-exist within the same module, as long as their pre-condition differed.

Though Acceleo 4 will allow multiple templates with the same signature to exist in a module, only the last one will ever be resolved for a call or override, all subsequent being ignored.

10.2.3. Query

Java services

The best practice for Acceleo 3 was to wrap java services inside of a query. The special `invoke` service was used to this end.

```
[query public hasStereotype(element : uml::Element, stereotypeName : String) : Boolean
=
    invoke('org.eclipse.acceleo.module.sample.services.UMLServices',
'hasStereotype(org.eclipse.uml2.uml.Element, java.lang.String)', Sequence{element,
stereotypeName})
/]
```

Acceleo 4 allows users to directly import java services.

10.2.4. File Block

The second argument for the file block in Acceleo 3 was a boolean (false = overwrite the file, true = append at the end of the file).

Acceleo 4 uses an enumeration, which allows the keyword "overwrite", "append" or "create" instead.

10.2.5. For Block

Simplified Syntax

Acceleo 3 allowed for loops in two formats:

```
[for (i : E | expr)]...[/for]
```

```
[for (expr)]...[/for]
```

In the second case, the loop variable was `self`.

Acceleo 4 only allows for the first of these two formats. Note that typing the iteration variable is optional in Acceleo 4.

Iteration count

Acceleo 3 defined an implicit variable, `i`, that held the current iteration count.

```
[for (feature : ecore::EStructuralFeature | class.eStructuralFeatures)]
iteration number [i/]
[/for]
```

Acceleo 4 does not define any similar variable.

before, separator, after

Acceleo 3 allowed users to specify a `before` expression that would be inserted right before the content generated by the loop body *if the loop had any iteration*. An `after` that would similarly

inserted after the loop body if it generated any content, and finally a `separator` which content would be inserted in-between each iteration result.

```
[for (number : Integer | Sequence{1, 2, 3}) before ('int[] array = new int[')
separator (', ') after ('];')][number/][/for]
```

Acceleo 4 only supports `separator`. This is not supported by the migration tool. To translate that in Acceleo 4 you could convert `before` and `after` into a `Let` holding the content of the iteration expression, then a `If` only generating before and after if the collection is not empty.

Pre-condition

Acceleo 3 allowed users to specify a pre-condition that would be evaluated every iteration and that would prevent all generation for that iteration if `false`.

```
[for (number : Integer | Sequence{1, 2, 3}) ? (isEven(number))]
[number/]
[/for]
```

Acceleo 4 does not have pre-conditions. This could be manually converted to an `If` at the start of the for body (this is not supported by the migration tool).

Init block

Acceleo 3 allowed variable initialization blocks on the for:

```
[for (feature : ecore::EStructuralFeature | class.eStructuralFeatures) {var : String =
'string'; className : String = class.name;}]
[number/]
[/for]
```

Acceleo 4 removes the initialization blocks altogether, so the variable declarations must be manually transformed into a `let` block before the `For` block (this is not supported by the migration tool).

Note the initialization block was evaluated before the for itself, and not for every loop.

Ranges

Acceleo 4 does not support ranges, e.g.:

```
[for (item : Integer | Sequence{1..5})]
[item/]
[/for]
```

Such for blocks are ignored by the migration tool.

10.2.6. Let Statement

Acceleo 3 only allowed a single variable per `Let`, forcing users to have multiple nested `Let` blocks to define more. The migration could aggregate multiple nested blocks into one with multiple variables if the nested blocks don't use one of the outer `Let's variable.

10.2.7. ElseLet Blocks

Acceleo 4 does not support elselet blocks, they are ignored by the migration tool.

10.2.8. Invocation

Acceleo 3 made use of implicit variables allowing module writers to avoid always specifying the target of an expression or call:

```
[template public generate(class : ecore::EClass)]
[name/] is equivalent to [class.name/] or [self.name/]
[eAllContents()/] is equivalent to [class.eAllContents()/] or [self.eAllContents()/]
[/template]
```

The implicit variable is always `self`, but the value of `self` may not be intuitive in all cases.

The migration replaces the implicit variable with the correct variable for Acceleo 4.

Template

Acceleo 3 : `self` is the first argument of the template.

Acceleo 4 : The migration tool makes the variable explicit, using the first argument of the template

Query

Acceleo 3 : `self` is the first argument of the query.

Acceleo 4 : The migration tool makes the variable explicit, using the first argument of the query

For

Acceleo 3 : `self` has the same value as the iteration variable.

Acceleo 4 : The migration tool makes the variable explicit, using the iteration variable

If

Acceleo 3 : The value of `self` is not changed within the `if` scope and remains the value of `self` outside of the `if`.

Acceleo 4 : The migration tool makes the variable explicit, using the parent context

let

Acceleo 3 : The value of `self` is not changed within the `let` scope and remains the value of `self` outside of the `let`.

Acceleo 4 : The migration tool makes the variable explicit, using the parent context

Expression

Acceleo 3/OCL : The value of `self` is defined by the current Acceleo scope and will not be altered by OCL.

Acceleo 4/AQL : The migration tool makes the variable explicit, using the parent context

You can have a look at the MTL to AQL migration guide.

10.2.9. Module Element Call

Template invocation

Acceleo 3 allowed special template calls such as the following:

```
[template public aTemplate()]
    [anotherTemplate() before ('inserted before generated body') separator ('in-
between') after ('inserted after generated body')/]
[/template]

[template protected anotherTemplate()]
    generated body
[/template]
```

Both `before` and `after` expression are handled by Acceleo 3 and will respectively generate their content before and after the callee's generated text. This is true even if the callee does not generate any text.

`separator` is not implemented by the engine so the migration strips it entirely.

Query invocation

Similar to template invocations, query invocations support `before`, `separator` and `after` expressions. None of which is implemented in the Acceleo 3 generation engine so they are stripped entirely.

10.2.10. Variable

Acceleo 3 supported unqualified type names for the variables.

```
[let var : EPackage = anotherVar.eContainer()]
    output text for EPackage named [var.name/]
[/let]
```

Acceleo 4 only accepts qualified types for the classifiers and the above becomes:

```
[let var : ecore::EPackage = anotherVar.eContainer()]
    output text for EPackage named [var.name/]
[/let]
```

10.2.11. Expressions

Acceleo 3 was using OCL as the underlying expression language, while Acceleo 4 is using AQL. Please look at the AQL Documentation for more information on migrating OCL expressions to AQL.

10.2.12. Set and Bags

The OCL collection types Set and Bag are not anymore available in AQL, which supports only two types: Sequence and OrderedSet. All collections are ordered. Thus the migration tool translates each Set into an OrderedSet and each Bag into a Sequence.

10.3. Behavior Changes

10.3.1. Modules

inheritance behavior

In Acceleo 3, once an overriding module (child) called a public or protected template of its extended module (parent), the execution flow would never come down to the child again until we `returned` out of the callee. This is contrary to other Object-oriented languages in which a `super` template could call down an `overriden` other template from the child when necessary.

If you consider the following simplified modules:

```
[module parentModule()/]

[template public aTemplate()]
    [anotherTemplate()/]
[/template]

[template protected anotherTemplate()]
    parent behavior
[/template]
```

```
[module childModule() extends parentModule/]

[template public main()]
    [aTemplate()/]
[/template]

[template protected anotherTemplate()]
    child behavior
[/template]
```

10.3.2. Query

Validation

In Acceleo 3, the return type of a query was not validated at compile time, so it was very easy for ClassCastExceptions to occur at runtime or for invalid templates to be written with the error only detected at runtime.

For example, the following will fail when we try to generate, but is valid for the compiler:

```
[template public generate(c : ecore::EClass)]
    [file (c.name.concat('.java'), false, 'UTF-8')]
    [for (attribute : ecore::EAttribute | getFeatures(c))]
        attribute name : [attribute.name/] [if (attribute.iD)]is id attribute[/if]
    [/for]
    [/file]
[/template]

[query private getFeatures(c : ecore::EClass) : Set(ecore::EAttribute) =
c.eStructuralFeatures/]
```

The template expected "getFeatures" to return a Set of Attributes, but the actual type is a set of EStructuralFeature. This will fail as soon as we try to generate for a class containing both attributes and references.

Acceleo 4 validates the return type of the query's body expression.

Cache

The MTL specification enforces that "A query is required to produce the same result each time it is invoked with the same arguments.". The result of a query call was thus cached in Acceleo 3, and never reevaluated. (This behavior could be disabled through a preference for Acceleo 3.)

Acceleo 4 will always reevaluate the query's body even if the same argument list is passed twice.

10.3.3. Let Statement

The Acceleo 3 let statement was equivalent to an "instance of" condition check to enter a block.

If we consider the following let block:

```
[let var : EPackage = anotherVar.eContainer()]
    output text for EPackage named [var.name/]
[/let]
```

In Acceleo 3, if the result of evaluating `anotherVar.eContainer()` is of type `EPackage` (the declared type of variable `var`), then this block will output the result of evaluating its body. In any other event, this would output no text and cause no failure as the block would be simply ignored if the types do not match.

In Acceleo 4, this same let block will cause validation errors if the type of `anotherVar.eContainer()` cannot be an `EPackage`.

10.4. Limitations

10.4.1. Comments

Comments are mostly ignored by the migration tool, except for module / template / queries documentation.