Generator User Guide Demo2

Generator User Guide Demo 2

In this demo we will again generate a Java Swing application, but unlike in previous Demo 1, this time it is going to be a single Java application per input model (in Demo 1 we generated a separate Java application for each input XML Document). The single Java application that Demo 2 creates, will contain all components mentioned in all XML Documents of the input model.

New Language

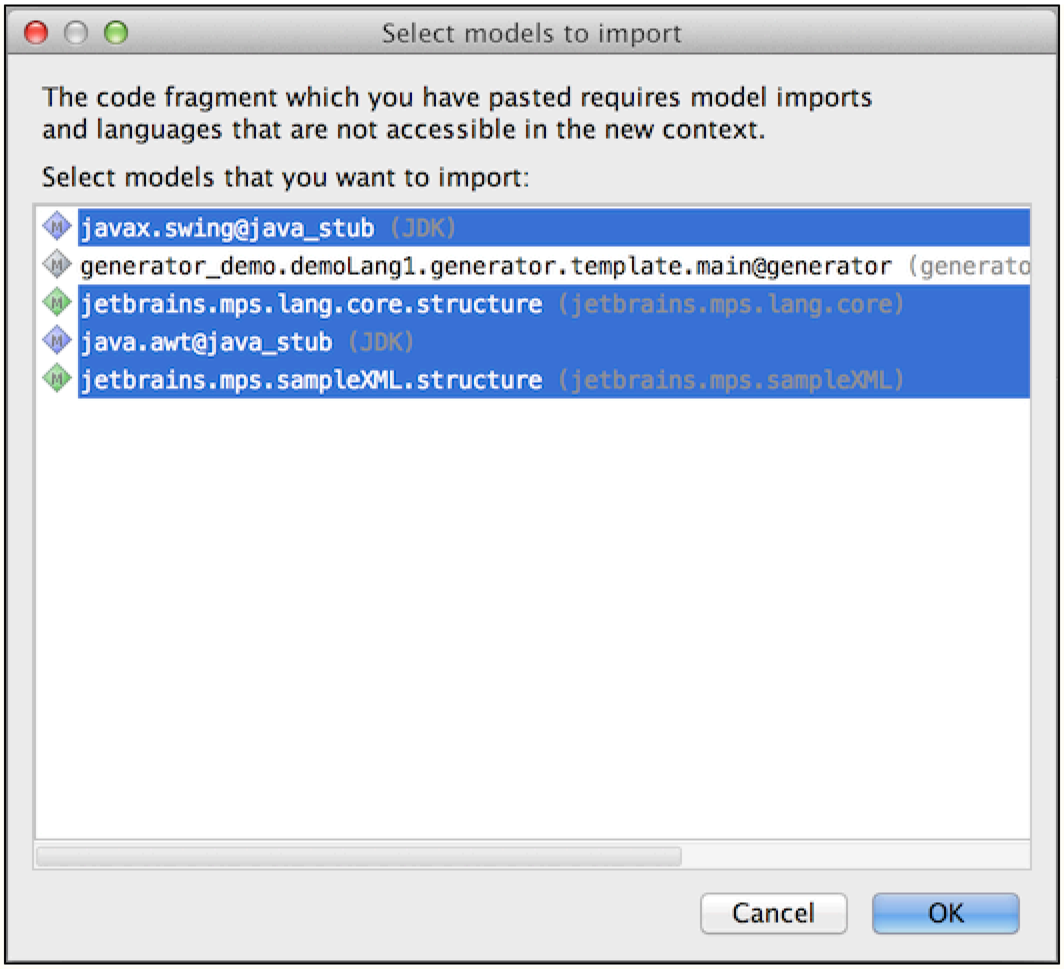
We need to do some little technical setup first:

- create a new language: 'generator_demo.demoLang2'
- in the language properties dialog add an extends dependency on 'jetbrains.mps.sampleXML' and 'jetbrains.mps.baseLanguage'
- create a new generator for this language, if it has not been created automatically (see Demo 1 for details).

As the code that we are going to generate is almost identical to that in Demo 1, we will copy the application class template from the demoLang1 generator:

- go to model 'main@generator' in demoLang1
- select the 'map_Document' template in project tree and copy it to clipboard
- return to model 'main@generator' in demoLang2 (select this model in tree) and paste from clipboard

> importing model on copy-paste
> The 'Imports and Languages' dialog will offer to import some additional models into our new demoLang2 generator model.
> Accept all but model 'main@generator' in demoLang1 (our new generator is not going to depend on the demoLang1 generator):
>
> - uncheck the demoLang1 generator model
> - press OK



The code fragment which you have pasted requires model imports and languages that are not accessible in the new context.

Select models that you want to import:

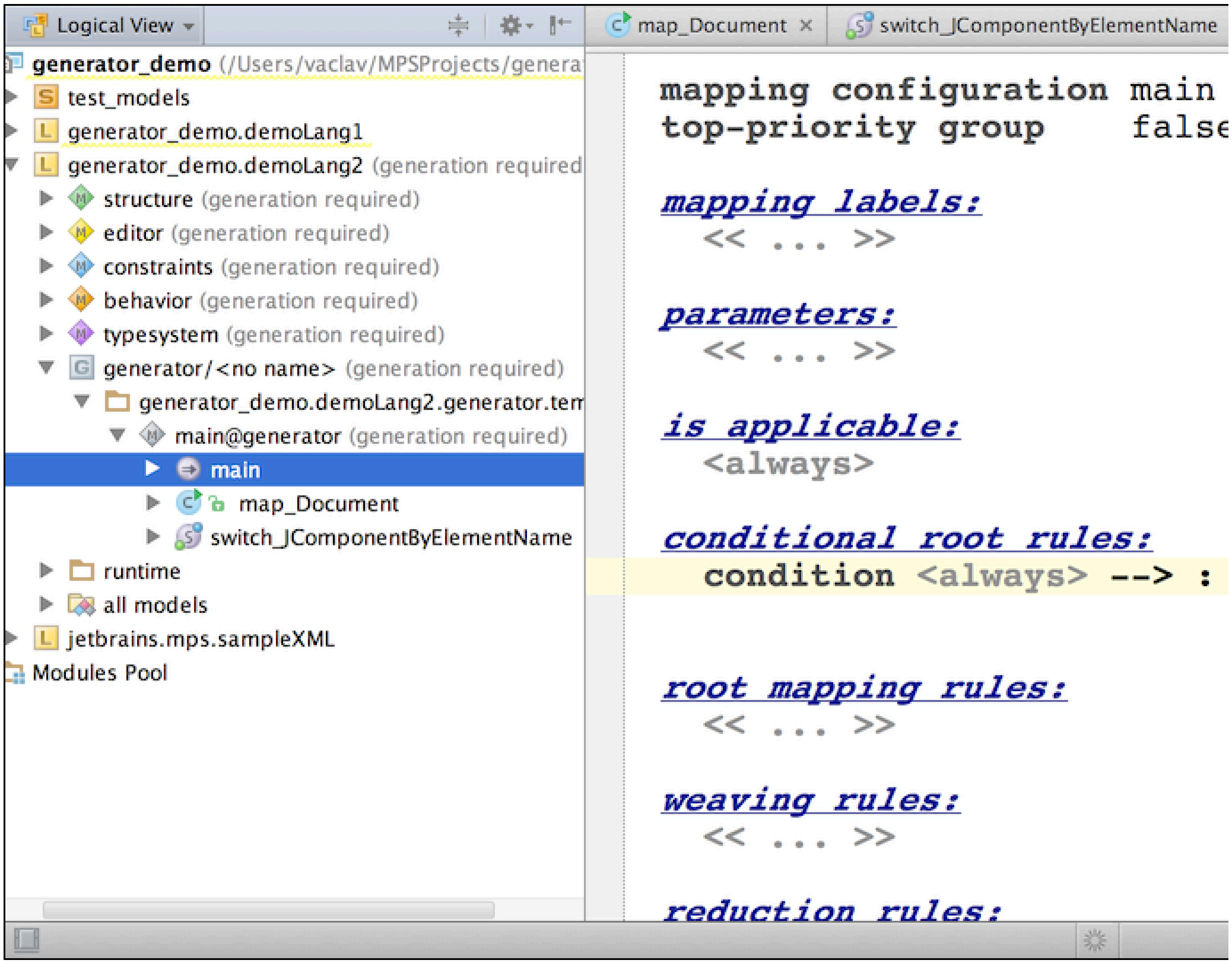
- javax.swing@java_stub (JDK)
- generator_demo.demoLang1.generator.template.main@generator (generato
- jetbrains.mps.lang.core.structure (jetbrains.mps.lang.core)
- java.awt@java_stub (JDK)
- jetbrains.mps.sampleXML.structure (jetbrains.mps.sampleXML)

- in the same manner copy the template switch 'switch_JComponentByElementName' from demoLang1 to the demoLang2 generator model

Conditional Root Rule

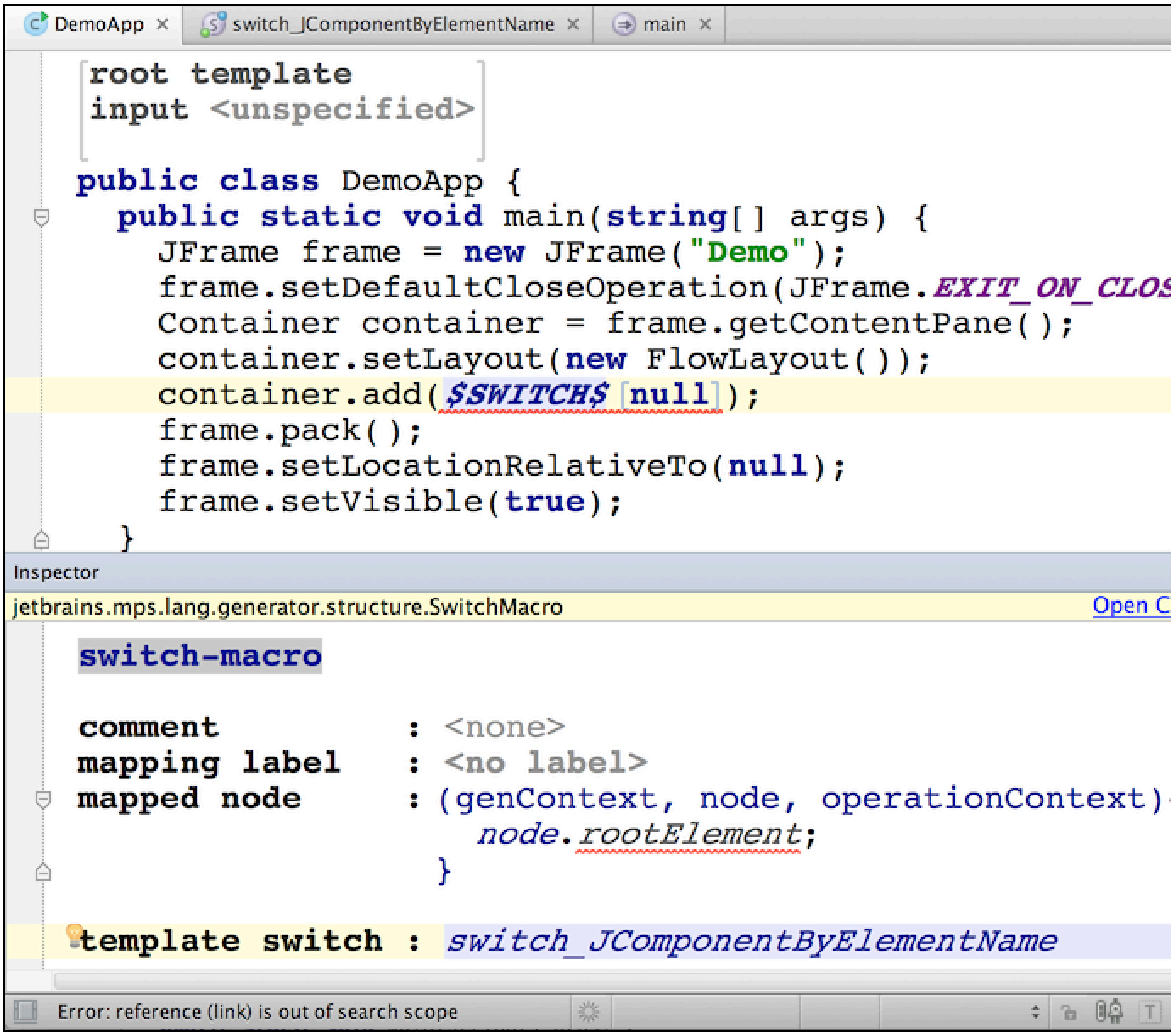
The process of code generation in MPS can be viewed as a series of model-to-model transformations. An AST in one language is gradually transformed into an AST expressed in concepts from another language. In each step individual concepts from one language get translated (reduced) into concepts from another language. Conditional Root Rule is somewhat special in that it creates new nodes out of nothing. It generates a brand new root node in the output model using a template referred to in this rule without having any original node in the input model.

- go to demoLang2 generator and open the mapping configuration 'main' in editor
- add new Conditional Root Rule (press Insert or Enter while cursor is in conditional root rules section)
- make reference to the 'map_Document' template so that this template is used to generate the resulting class


Logical View
map_Document
switch_JComponentByElementName
generator_demo (/Users/vaclav/MPSProjects/genera
test_models
generator_demo.demoLang1
generator_demo.demoLang2 (generation required
structure (generation required)
editor (generation required)
constraints (generation required)
behavior (generation required)
typesystem (generation required)
generator/<no name> (generation required)
generator_demo.demoLang2.generator.tem
main@generator (generation required)
main
map_Document
switch_JComponentByElementName
runtime
all models
jetbrains.mps.sampleXML
Modules Pool

```
mapping configuration main
top-priority group     false

mapping labels:
  << ... >>

parameters:
  << ... >>

is applicable:
  <always>

conditional root rules:
  condition <always> --> :


root mapping rules:
  << ... >>

weaving rules:
  << ... >>

reduction rules:
```

- open the 'map_Document' template (Control + <left-click> or Cmd + <left-click> on the reference in the editor) so we can tweak it a bit for it to work in the changed context



```
root template
input <unspecified>

public class DemoApp {
  public static void main(string[] args) {
    JFrame frame = new JFrame("Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOS
    Container container = frame.getContentPane();
    container.setLayout(new FlowLayout());
    container.add($SWITCH$[null]);
    frame.pack();
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
  }
```

```
jetbrains.mps.lang.generator.structure.SwitchMacro

switch-macro

comment          : <none>
mapping label    : <no label>
mapped node      : (genContext, node, operationContext)
                     node.rootElement;
                   }

template switch  : switch_JComponentByElementName
```

```
Error: reference (link) is out of search scope
```

- in the template's header remove the mention of Document - press Del while cursor is on the word 'Document' (we are using this template in Conditional Root Rule, which no longer passed a Document as the template's input)
- remove the property-macro from the class name, as now we only get a single class, we can hard-code its name
- rename the generated class to 'DemoApp'
- open the Inspector for the SWITCH macro and resolve the reference to the template_switch by hitting Control + Space

LOOP-macro

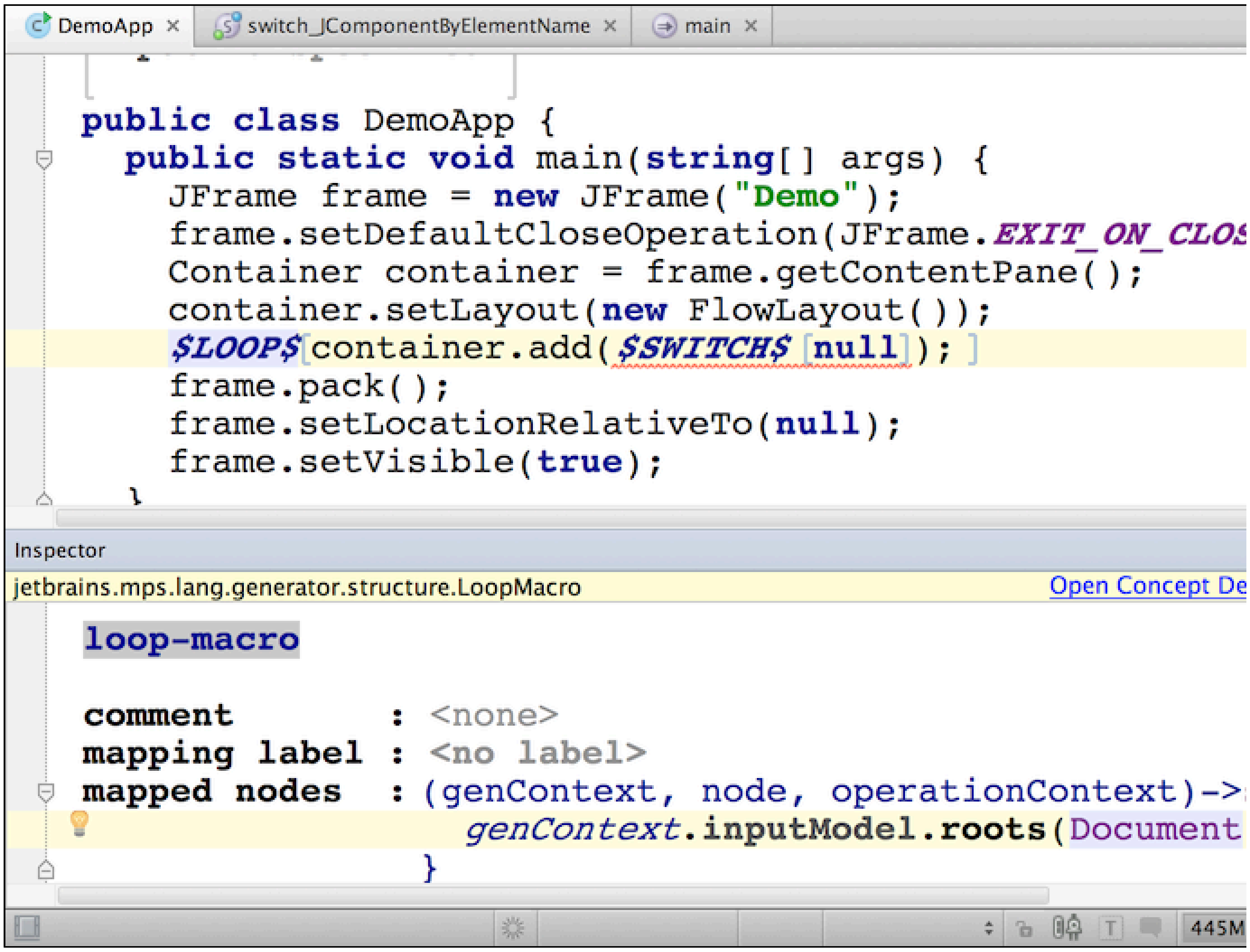
The SWITCH-macro in 'DemoApp' template is expecting an XML Document on the input, since this is how we implemented it in Demo 1. It then extracts the Document's root element and applies a template switch to it. However, there may be multiple Documents in the input model (in fact in our demo app will have two - a button and a label), so we need to gradually process them all in sequence and generate an appropriate "container.add(null);" statement for each of these Documents.
To provide the SWITCH-macro with a single Document node, we will wrap the 'container.add(..);' statement inside a LOOP-macro:

- in the template code select the statement 'container.add($SWITCH$[null]);' (with your cursor placed inside the statement, use Ctrl/Cmd+W to expand block selection)

> ⓘ make sure that +whole statement+ is 'wrapped', i.e. including the ';' symbol

- press Ctrl-Shift+M to add node-macro

- choose $LOOP$ from the auto-completion menu (Ctrl+Space)
- enter code in the LOOP's mapped nodes function as below



```
public class DemoApp {
  public static void main(string[] args) {
    JFrame frame = new JFrame("Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOS
    Container container = frame.getContentPane();
    container.setLayout(new FlowLayout());
    $LOOP$[container.add($SWITCH$ [null]); ]
    frame.pack();
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
```

```
jetbrains.mps.lang.generator.structure.LoopMacro

loop-macro

comment       : <none>
mapping label : <no label>
mapped nodes  : (genContext, node, operationContext)->
                  genContext.inputModel.roots(Document
                }
```

The mapped nodes function will find all Documents in the input model and return a sequence of these Documents to the generator (the return statement can be omitted in BaseLanguage).
Then, the generator will iterate through this sequence, create 'container.add(..);' statement for each Document and gradually pass each Document as an input node to the SWITCH-macro.

That's it. The last step, as usual, is hitting Shift-F9 to make the generator.

Running First Test

To test the demoLang2 generator we need the exactly same input as in Demo 1 except that we are going to 'engage' different language on its generation.

- go to the 'test1' model in 'test_models' solution (select it in project tree)
- choose Clone Model in the popup menu
- enter the new model name: 'generator_demo.test2'
- in properties dialog of the 'test2' model, in the Languages Engaged on Generation section:
  - remove 'generator_demo.L1' language
  - add 'generator_demo.L2' language

Generate the 'test2' model and preview the generated sources.

Abandon Root Rule

Note that the generated output also contains two 'generated' XML files: Button.xml and Label.xml. MPS did not generate those

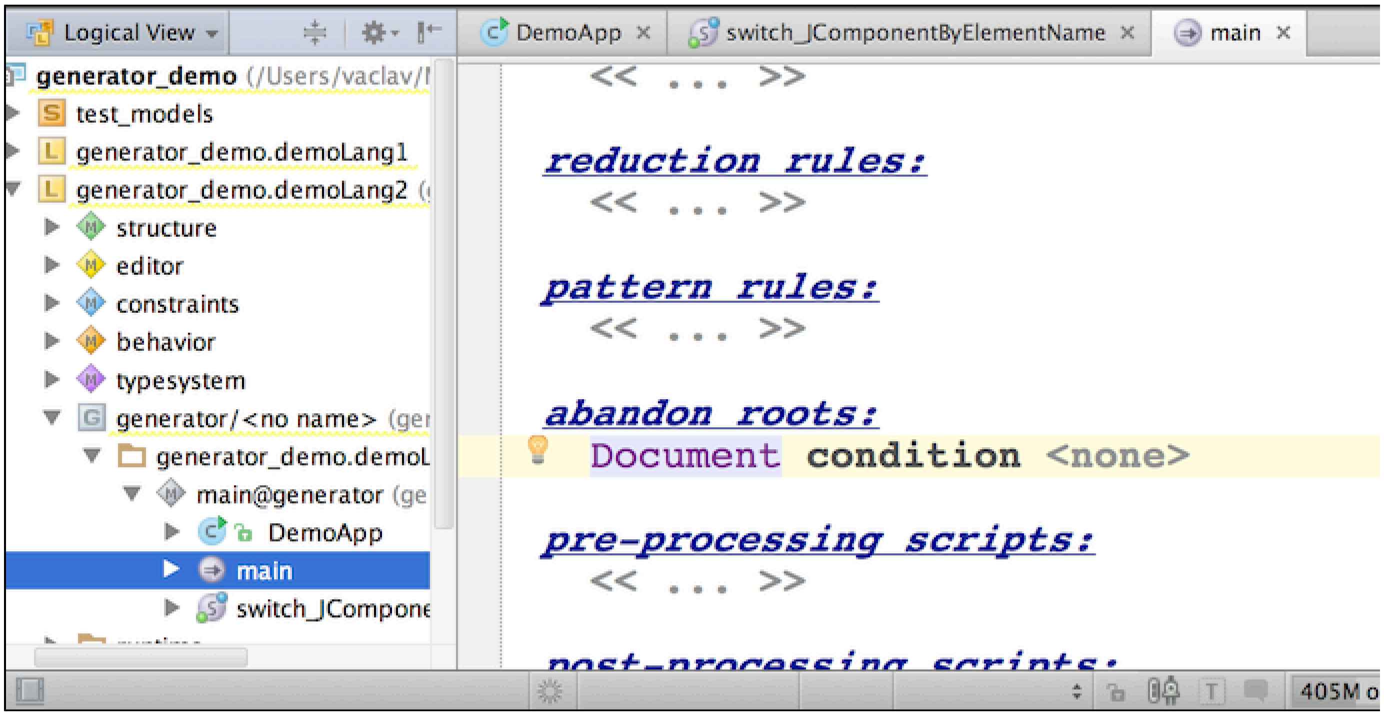
XML files in previous Demo 1. Why does it do so in this demo?

Because in Demo 1 we have defined explicit mapping of input Documents to output Classes using Root Mapping Rule, the input was consumed and the xml files were not generated..
In Demo 2 we do not explicitly map input Documents to any output roots and so they remain in the model until the text-generation phase, which turns them into xml files.

If an input root node is not explicitly mapped into an output root node, then MPS will copy this input root node to output model.

To prevent copying Documents to the output model we will use the Abandon Root Rule.

- return to demoLang2 generator and open mapping configuration 'main' in editor
- add a new Abandon Root Rule - press Insert while cursor is in the abandon roots section
- choose 'Document' as the concept for application of the rule



```
<< ... >>

reduction rules:
  << ... >>

pattern rules:
  << ... >>

abandon roots:
  Document condition <none>

pre-processing scripts:
  << ... >>

post-processing scripts:
```

After rebuilding the project you will no longer get xml files on the output.