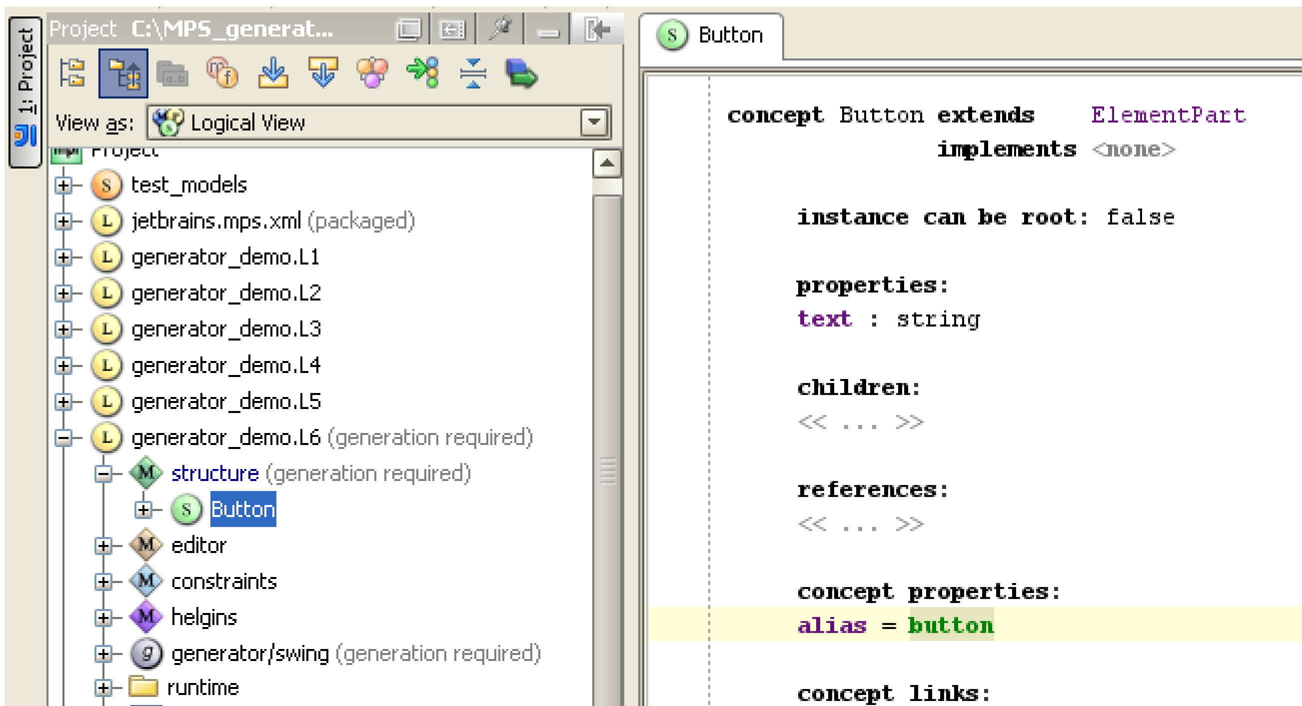
Generator User Guide Demo6

Generator User Guide Demo 6

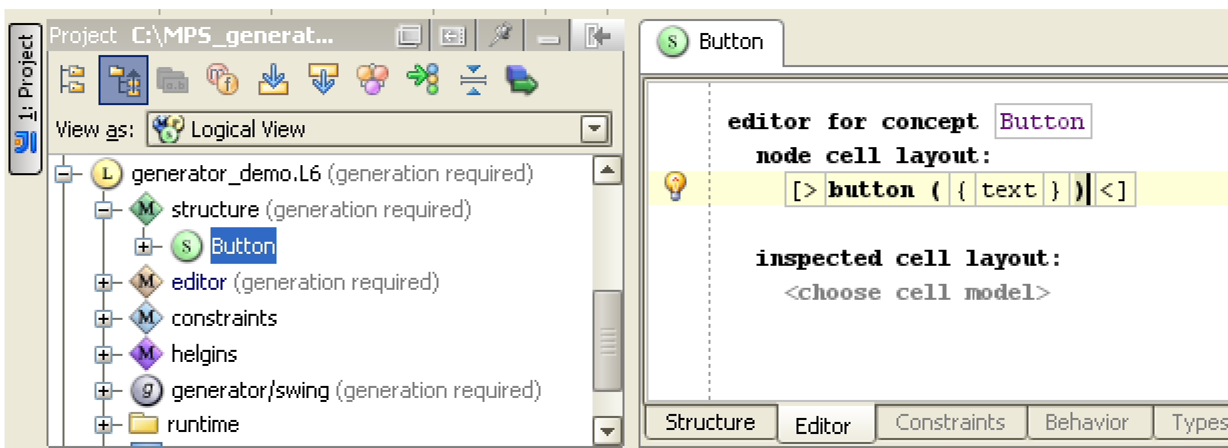
This final demo is probably most exciting of all demos, because here we are going to create a 'real' language in the sense that this language will actually define couple of higher-level concepts.
We will also see how easy this DSL is integrated with existing languages - 'jetbrains.mps.sampleXML' and 'generator_demo.L5' (created earlier in the Demo 5).

New Language

- create new language 'generator_demo.L6'
- in language properties dialog add extended language : 'jetbrains.mps.sampleXML'
- create new generator for this language (see Demo 1 for details)
- in L6 structure model create new concept 'Button' extending concept 'ElementPart' (from 'jetbrains.mps.sampleXML')
- in concept 'Button' declare property 'text' and add alias (we need alias to make auto-completion menu look nice):



- create editor for the 'Button' concept:



The editor consists of three cells: two constant-cells (shown in bold) and one property-cell (shown as `{text}`) which will render

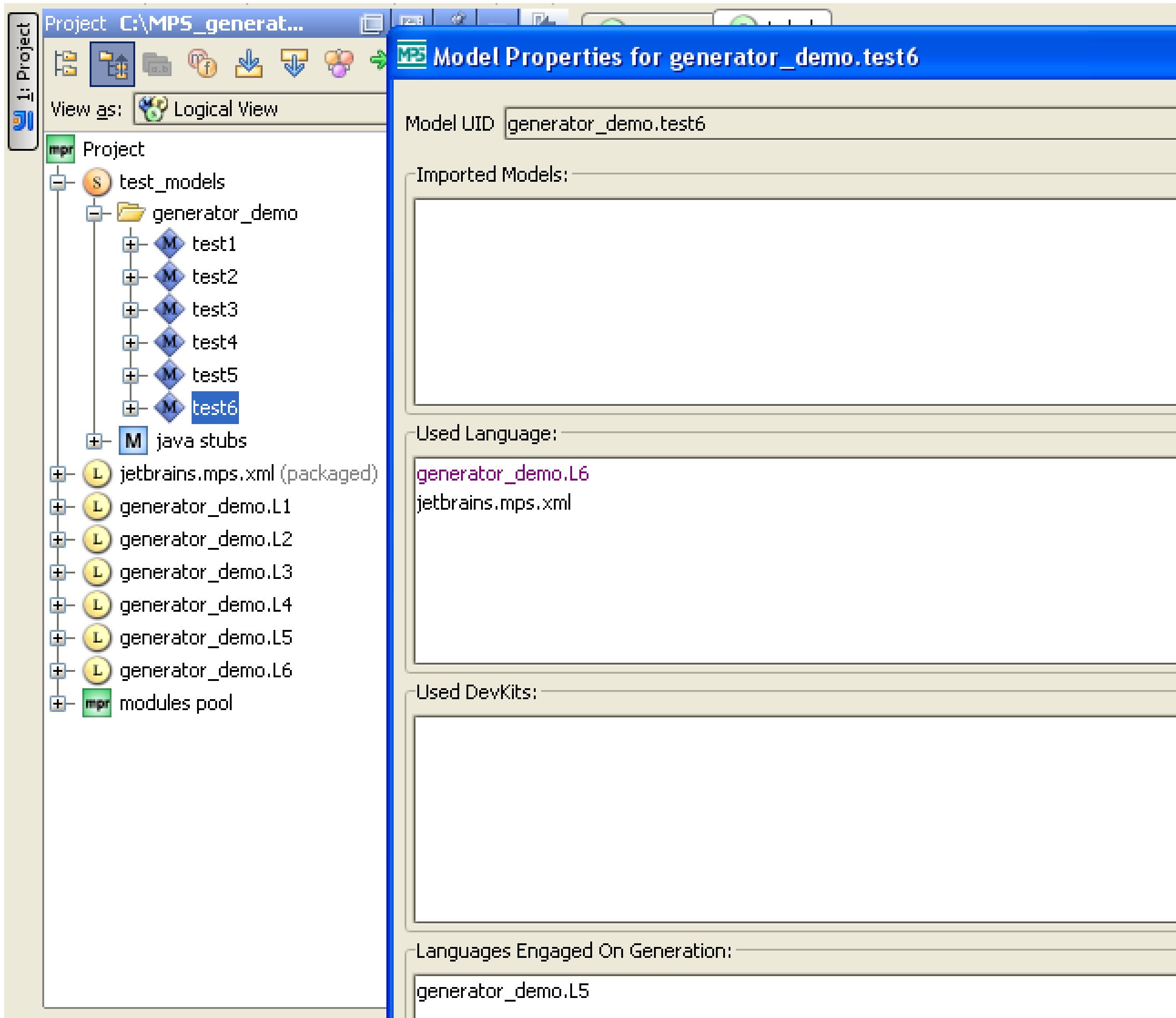
value of the 'text' property.

- create similar concept - 'Label'
- re-generate language (Ctrl-F9)

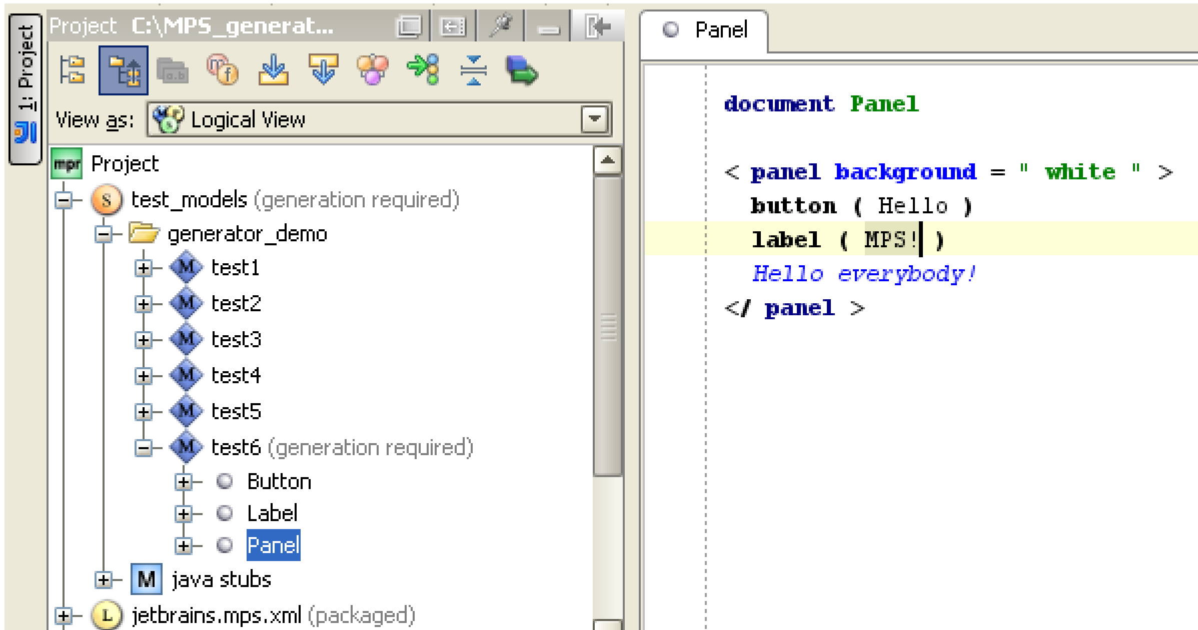
New Test Model

Now let's try out our new DSL:

- go to the 'test_models' solution
- clone model 'test5' to model 'test6' (this time DO NOT replace engage on generation language generator_demo.L5 -> generator_demo.L6)
- add the 'generator_demo.L6' language to the used languages section (new!)



- in model 'test6' open document 'Panel' and replace 'label' elements with our new button and label:



With that new DSL our 'code' became clearlier, shorter and less error prone (for the instance, users can't add elements inside labels any more).

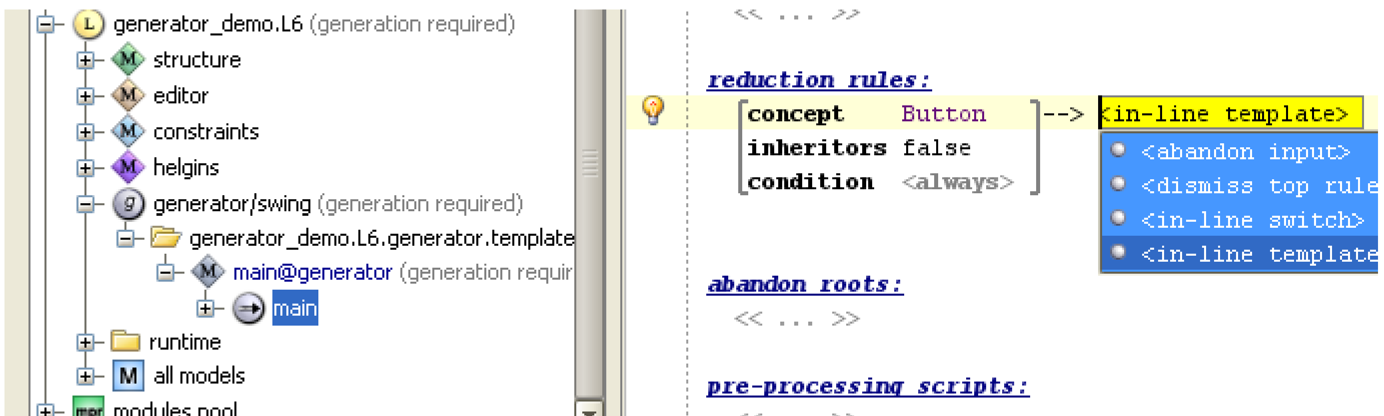
Generator

We could generate methods and method calls out of button and label in the pretty same manner as we have it done earlier for XML elements.
But this isn't a good idea, because this way we will introduce indirect dependency between L6 generator and L5 generator (L6 generator would know about implementation details of L5 generator).

Luckily we have another option.
As we are on a higher level of abstraction now (comparing to XML) we probably can simply reduce our semantically rich concepts to lower-level concepts (XML elements) and don't care any more of who and how will transform those concepts into yet more primitive concepts.

Reduction Rules

- open (empty) mapping configuration 'main' in L6 generator
- add new reduction rule applicable to concept Button
- choose in-line template as the rule consequence:



The template is going to be simple and it will not require any 'context' so it doesn't make a much sense to create an external template as we did in other demos.

- choose Element as template's content node:

```
reduction rules:
  [concept     Button    ]--> <T Ele              T>
  [inheritors false      ]      Element        concept instance
  [condition   <always>  ]      ElementPart    concept instance
```

- create 'button' XML element with 'text' attribute
- attach property macro to the attribute value
- enter code in the macro's value function as shown:

```
reduction rules:
  [concept     Button    ]--> <T  < button text = " $[text]" >  T>
  [inheritors false      ]            ...
  [condition   <always>  ]        </ button >

abandon roots:
  << ... >>

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

Inspector
  property macro

  value : (node, templateValue, genContext, operationContext)->S
            node.text;
          }
```

That's it. Our high-level button is reduced to a 'button' element with attribute 'text', which will presumably be transformed to java by somebody else.

- add similar reduction rule but applicable to concept Label

First Test (Failure)

Re-generate generator and generate model 'test6'.

There will be a bunch of error messages in MPS messages view - one of weaving rule has failed.
Click on 'was input node' error message. MPS will show you node to which generator was trying to apply the rule:

You can also click on 'was template' message to find out which rule has failed:

This is weaving rule in L5 generator (we are generating 'test6' model with two generators: L5 and L6) and it has failed to find weaving context node by mapping label 'main_class'.

The 'main_class' label is assigned to 'DemoApp' class by conditional root rule (visible on the screenshot above) so what's wrong? Why we couldn't find the 'DempApp' by that label?

To answer those questions let's take a closer look at the generation process.

At first, generator creates a transient model which will serve as an output model. This model name is 'generator_demo.test6@2_1'.

Then generator applies conditional root rule and creates class 'DemoApp' in output model.

Then, at some point, the 'weave_Panel' template is invoked and its LOOP-macro is executed (we can see //add children and other familiar statements). The LOOP-macro creates statement 'component.add()' for each child of 'panel' element and reduces child to generate an actual argument in the 'add()' method call.

Our 'panel' element contains two children - button and label. Both of them are high-level abstractions and their reduction yields corresponding XML elements. (We can see it on first screenshot where XML elements are passed as actual argument to method call).

This ends generation of transient model 'generator_demo.test6@2_1' and generator creates new transient model 'generator_demo.test6@2_2'. This model becomes new output model and former output model becomes new input model.

Generator starts to generate new input model 'generator_demo.test6@2_2' pretending that there was nothing before.
Generator has no memory about previous activities with only one exception - conditional root rules are never applied twice.
Thus, this time, the output 'DemoApp' class is created by copying of 'DemoApp' class from input model, not by applying of a co nditional root rule.
As result the 'DemoApp' class in model 'generator_demo.test6@2_2' is not any more associated with mapping label 'main_class '.

After that generator detects that there are 'button' and 'label' XML elements in input model, tries to apply rules to them and crushes.

Now that we understand what is happening, the next question is how to fix it.

We are relying on L5 generator and we know that L5 generator works well providing that input model is a 'valid XML' (we have tested it in Demo 5).
In Demo 6 our input model is not exactly a 'valid XML'. Moreover, we witnessed that the transient model 'generator_demo.test6@2_1' is not a valid model in any sense - a method call can't accept an XML element as its actual argument (you can even find warning massage "child 'Expression' is expected for role 'actualArgument' but was 'Element'" in MPS message view).

Now we could probably return to L5 generator to make some 'improvements' to allow it to handle 'invalid' input models.
Fortunately, there is better option - we can divide the generation process in two steps so that the original input model will be transformed into a 'valid XML' model on first step, and then this 'valid XML' model will be transformed to Java by L5 generator on second step.

Dividing Generation Process into Steps

We will get 'valid XML' from input model 'test6' if we reduce button and label (in the 'Panel' document) into XML elements.

This transformation must happen before L5 will start to transform XML into Java.
In other words, reduction rules specified in L6 generator must be applied before any rules in L5 generator.

Let's specify this constraint:

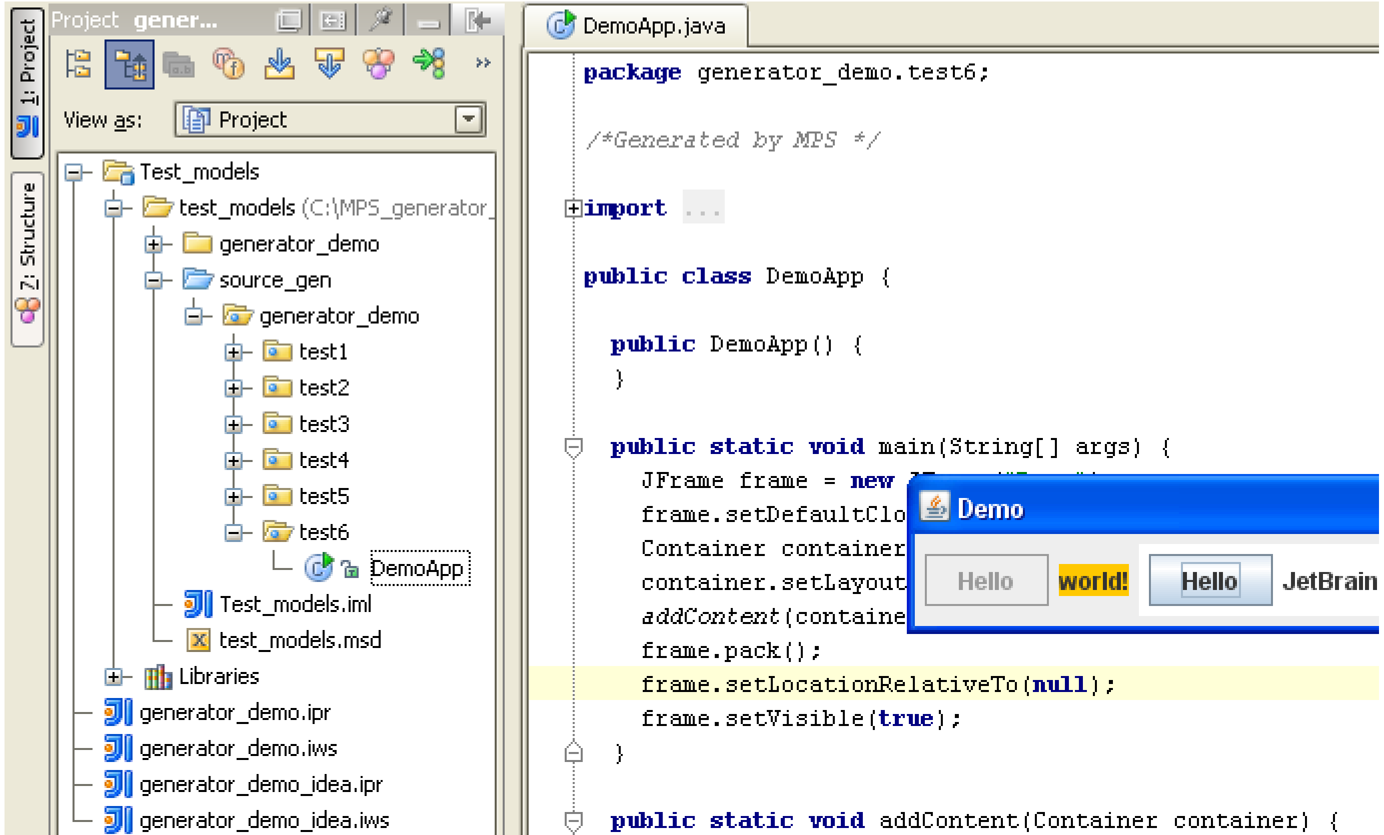
- go to generator in language 'generator_demo.L6' and open the generator properties dialog
- in depends on generators section add reference on L5 generator: 'generator_demo.L5/swing'
- in mapping constraints section add constraint statement as shown:

This constraint statement can be translated as:

> Execute all my mapping configurations before any mapping configurations in generator 'generator_demo.L5/swing'.

Second Test

Generate files from model 'test6' - should be no problems now, run the generated 'generator_demo.test6.DemoApp' class:

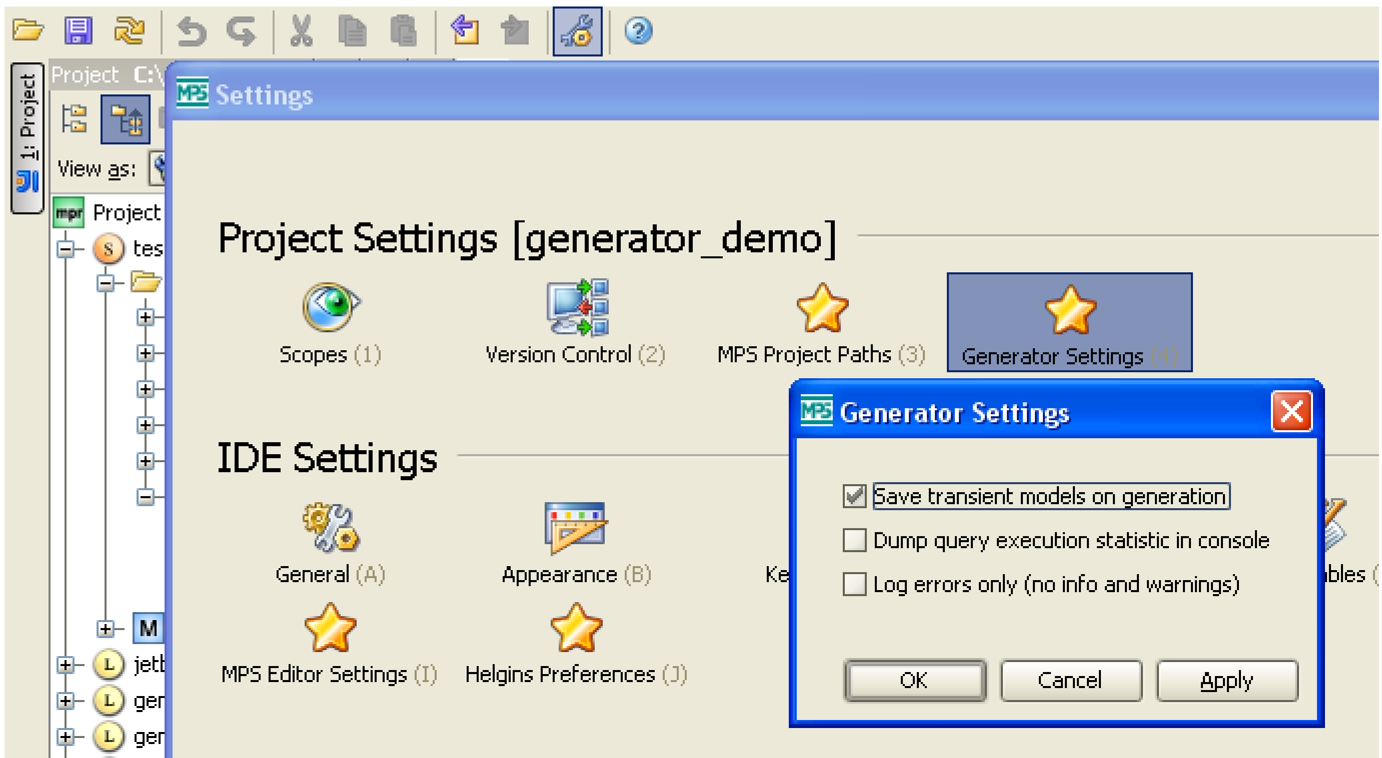


Saving Transient Models

Our L5 and L6 generators are working together well because we have divided the generation process into two steps.
On the first step L6 generator reduces our high-level concepts (button and label) into corresponding XML elements and produces 'valid XML' model as output.
On the second step L5 generator generates java from that 'valid XML' model.

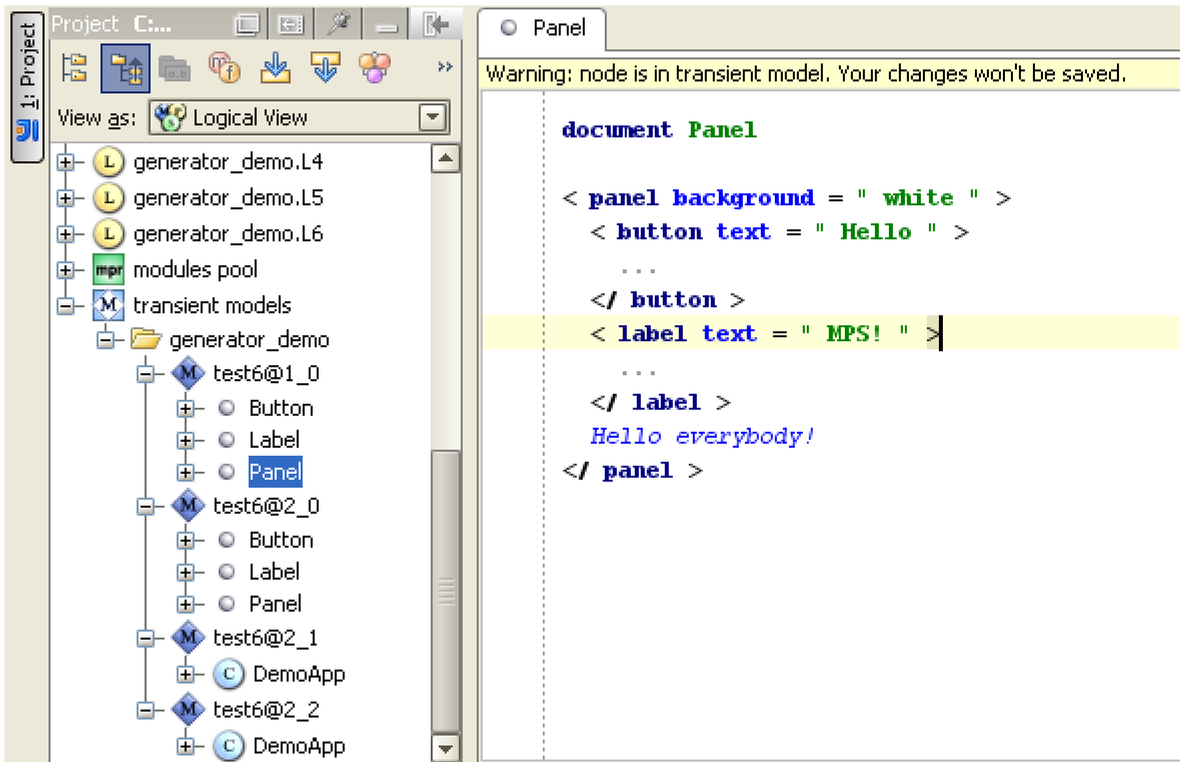
Thus there should be at least one transient model between steps 1 and 2.
Let's take a look, what models have actually been created in the course of generation:

- in Project Settings click on Generator Settings and check the Save transient models on generation option:



- generate model 'test6' in solution 'test_models'
- in project tree find and expand node named 'transient models' (this node is at the bottom)
  There are (surprisingly) four transient models.
  We can browse transient models and we will find sometimes subtle and sometimes dramatic differences between them.
  This will give us a clue about what kind of transformation has taken place on each step.

For the instance, open 'Panel' node in model 'test@1_0' (this model has been generated on very first step).



We can see that high-level button and label have been replaced with their XML counterparts, but text 'Hello everybody!' is still here.

In the next model 'test6@2_0' the text 'Hello everybody!' has been replaced with a 'label' element (as result of application of pre-processing script 'fix_text' in L5 generator).

Model 'test6@2_1' contains generated 'DemoApp' class.

And model 'test6@2_2' contains the same class but string "MPS!" is replaced with "JetBrains MPS!" (as result of application of post-processing script 'refine_text' in L5 generator).

Root Nodes Copying and Reduction Rules

As we can see, model 'test6@1_0' is identical to the original input model 'test6' except for high-level button and label, that have been reduced into XML elements.
This is what we wanted but it is probably still not clear, how roots in 'test6@1_0' has been created (we don't have any 'root rules' that create Document) and why our reduction rules have been applied.
From our previous experience with reduction rules we remember that we had to create a COPY_SRC-macro for reduction to take place. But in L6 generator we don't have SOPY_SRC-macros.

copying and reduction
Whenever generator meets input root node, for which none of root mapping rules or abandon root rules is applicable, generator creates a deep copy of that root in the output model.
While copying node, generator is always trying to find an applicable reduction rule.
If reduction rule is found, it is applied. Otherwise generator creates output node (instantiates the same concept), duplicates all properties and references of input node (local references are re-resolved) and repeats the same copying procedure for all children nodes recursively.

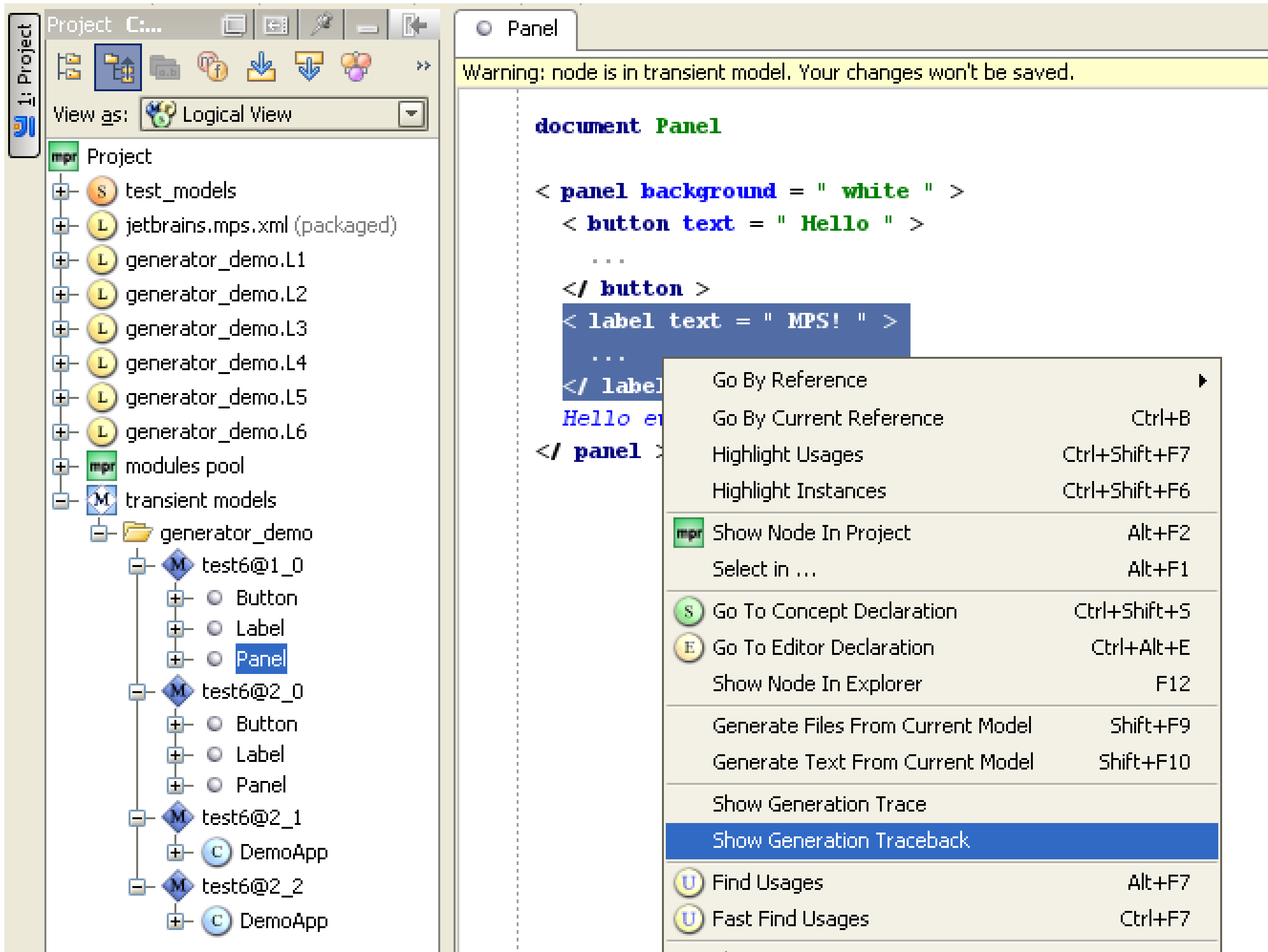
This way input documents (Button, Label and Panel) from model 'test6' has been copied to model 'test6@1_0' and reduction has been applied to button and label in the 'Panel' document.

Returning to COPY_SRC_macro - it doesn't 'invoke' a reduction (as it might seem). Instead it merely applies the same copying procedure to the mapped node and, if any reduction rules are happen to be applicable, then the transformation occurs.

Using Generation Tracer Tool

We already know how to save and browse transient models. Actually, with the option Save transient models on generation acti vated, MPS not only saves transient model but also collects a great deal of data regarding the process of transformation.
This data can be viewed using the Generation Tracer Tool.

For instance, open 'Panel' document in transient model 'test6@1_0', select 'label' and choose Show Generation Traceback in popup menu.



The Generation Tracer View will be opened.

In the root of this tree there is the 'label' node for which we have requested traceback info (blue arrow denotes an output node) and the rest of the tree shows the sequence of events in reversed order that led to this output to occur.

Looking at this tree and clicking on its nodes (to open in editor) we can reproduce the process of transformation is great details.

For instance, we can see that the output 'label' is created by template in 'reduce Label' rule.

... and the 'reduce Label' rule has been applied to input node 'Label' while copying of 'Panel' document which is root input node.

Project C:...

View as: Logical View

- modules pool
- transient models
  - generator_demo
    - test6@1_0
      - Button
      - Label
      - Panel
    - test6@2_0
      - Button
      - Label

Panel | main | Panel

```
document Panel

< panel background = " white " >
  button ( Hello )
  label ( MPS! )
  Hello everybody!
</ panel >
```

Generation Tracer label

```
label (r:88b505b2-987b-4b6e-a732-bf17140c0b74(generator_demo.test6@1_0))
  label (r:683fac40-9ea9-4460-b915-89b6339af13e(generator_demo.L6.generator.template.main@genera
    <in-line template> (r:683fac40-9ea9-4460-b915-89b6339af13e(generator_demo.L6.generator.temp
      reduce Label (r:683fac40-9ea9-4460-b915-89b6339af13e(generator_demo.L6.generator.templa
        <no name>[Label] (r:f3da1ffc-c8bb-4d02-9ae2-698dbf98beaa(generator_demo.test6))
          <copy>
            panel (r:f3da1ffc-c8bb-4d02-9ae2-698dbf98beaa(generator_demo.test6))
              <copy>
                Panel (r:f3da1ffc-c8bb-4d02-9ae2-698dbf98beaa(generator_demo.test6))
```

9: Changes | 3: Usages | 0: MPS Messages | 4: Output | Generation Tracer

End of Demo 6