

MPS Generators

Eugen Schindler

Heavily inspired by [the MPS generators user manual](#) and the [“building maintainable generators” guide](#) from Itemis (@coolya)

Index

- What are (MPS) generators?
- Model-to-model transformations
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- Model-to-text transformations
- Common generator patterns (partially inspired by Itemis guide)
- Basic dos and don'ts
- Further reading

Index

- **What are (MPS) generators?**
- Model-to-model transformations
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- Model-to-text transformations
- Common generator patterns
- Basic dos and don'ts
- Further reading

What are (MPS) generators?

- Part of a language specification that defines denotational semantics for the concepts of the language

Denotational semantics as source-to-source translation [\[edit\]](#)

It is often useful to translate one programming language into another. For example, a concurrent programming language might be translated into a [process calculus](#); a high-level programming language might be translated into byte-code. (Indeed, conventional denotational semantics can be seen as the interpretation of programming languages into the [internal language](#) of the category of domains.)

In this context, notions from denotational semantics, such as full abstraction, help to satisfy security concerns. [\[16\]\[17\]](#)

3.2 Term Rewriting Systems

Definition 3.26 (Term Rewriting System) A *Term Rewriting System* (TRS) is a pair (Σ, R) of an *alphabet* or *signature* Σ and a set of reduction rules (rewrite rules) R . The alphabet Σ consists of

1. a countably infinite set of *variables* x_1, x_2, \dots also denoted x, y, z, z', y', \dots
2. a non-empty set of *function symbols* or *operator symbols* F, G, \dots equipped with an *arity* (a natural number) which specifies the number of arguments it is supposed to have. Besides unary, binary, etc., function symbols we also have 0-ary, also called *constants*.

Definition 3.27 (Terms) The set $Terms(\Sigma)$ of terms over Σ is defined inductively:

1. all variables x, y, z of Σ belong to $Terms(\Sigma)$,
2. if F is a function symbol of Σ with arity n and $t_1, \dots, t_n \in Terms(\Sigma)$ then $F(t_1, \dots, t_n) \in Terms(\Sigma)$.

I would say: it's a specification as you see them typically in term rewrite systems

Definition 7.1 (Denotational Semantics) The denotational semantics of a sequential program is defined by induction on the structure of statements.

- a) $\mathcal{D}(x := e)\sigma_0 = \{(\sigma_0 : x \mapsto \mathcal{E}(e)\sigma_0)\}$,
- b) $\mathcal{D}(\text{skip})\sigma_0 = \{\sigma_0\}$,
- c) $\mathcal{D}(S_1 ; S_2)\sigma_0 = \{\sigma \mid \text{there exists a } \sigma_1 \text{ such that } \sigma_1 \in \mathcal{D}(S_1)\sigma_0 \text{ and } \sigma \in \mathcal{D}(S_2)\sigma_1\}$.

To simplify definitions, we define for functions $F_1, F_2 : STATE \rightarrow \wp(STATE)$ the function $SEQ(F_1, F_2) : STATE \rightarrow \wp(STATE)$ by

$$SEQ(F_1, F_2)\sigma_0 = \{\sigma \mid \text{there exists a } \sigma_1 \text{ such that } \sigma_1 \in F_1(\sigma_0) \text{ and } \sigma \in F_2(\sigma_1)\}.$$

Then we indeed have a compositional formulation:

$$\mathcal{D}(S_1 ; S_2) = SEQ(\mathcal{D}(S_1), \mathcal{D}(S_2)).$$

Intermezzo - semantic equivalence

Let $S_1 \sim S_2$ denote that S_1 and S_2 are semantically equivalent, that is, $\mathcal{D}(S_1) = \mathcal{D}(S_2)$. Note that \sim is an *equivalence relation*, that is, for all S, S_1, S_2, S_3 ,

(reflexive) $S \sim S$,

(symmetric) if $S_1 \sim S_2$ then $S_2 \sim S_1$

Wikipedia (2005)

syllabus "Berekeningsmodellen" TU/e (2002)

What are MPS generators?

- Dedicated aspect of a language
- Covers all model-to-model transformations
- MPS generator engine responsible for performing transformations specified in generators
- Model-to-text transformations (usually postponed until the last possible moment) officially covered by another aspect: textgen

Index

- What are (MPS) generators?
- **Model-to-model transformations**
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- Model-to-text transformations
- Common generator patterns
- Basic dos and don'ts
- Further reading

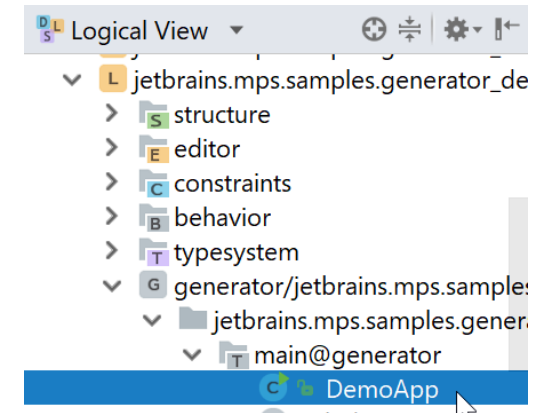
Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- Mapping labels
- Generation context
- The generator algorithm
- Mapping scripts

Model-to-model transformations

- Templates
 - **Types of templates & template fragments**
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- Mapping labels
- Generation context
- The generator algorithm
- Mapping scripts

Templates



- Transformations described by means of **templates** (in **template declarations**), which are written in the **output language** (i.e. the language of the output model)
- Applicability of individual templates is defined by **generator rules**, which are grouped into **mapping configurations**
- Root template (lives in a root node) → for producing a root node:

Input model

document Button

```
< button text = " Hello " enabled = " false " >
```

...

```
</ button >
```

written in: jetbrains.mps.sampleXML

Root template

```
root template
input Document

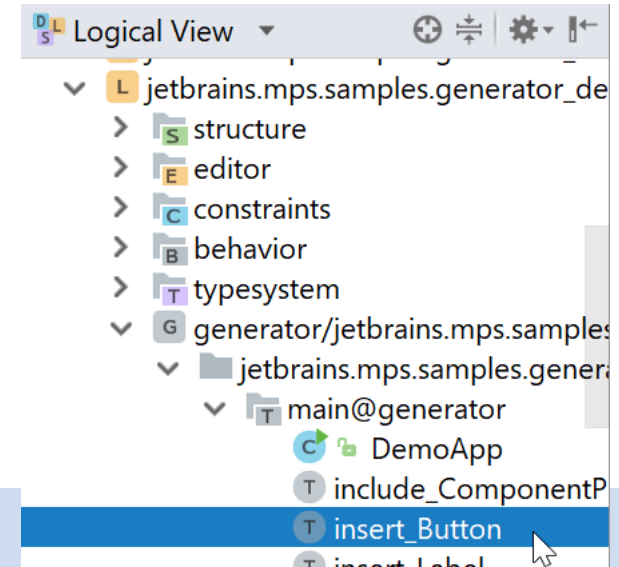
public class $node.name:string {
    public static void main(string[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        container.add($SWITCH$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

written in (language of output model): jetbrains.mps.baseLanguage (java)

Templates



- External template (lives in a root node) → for producing a non-root node:



Root template

```
template insert_Button
input Element
```

```
parameters
<< ... >>
```

```
content node:
```

```
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF method
```

```
public static
```

```
    Component $genContext.unique name from (templateValue) in context (<no node>) :string() {
        JButton component = new JButton();
        if( node.attribute.findFirst({~it => it.name.equals("text") }).isNotNull )<T
            component.setText("$node.attribute.findFirst({~it => it.name.equals("text") }).value");
        T>
        $INCLUDE$ { node } → include_ComponentProperties
        return component;
    }
```

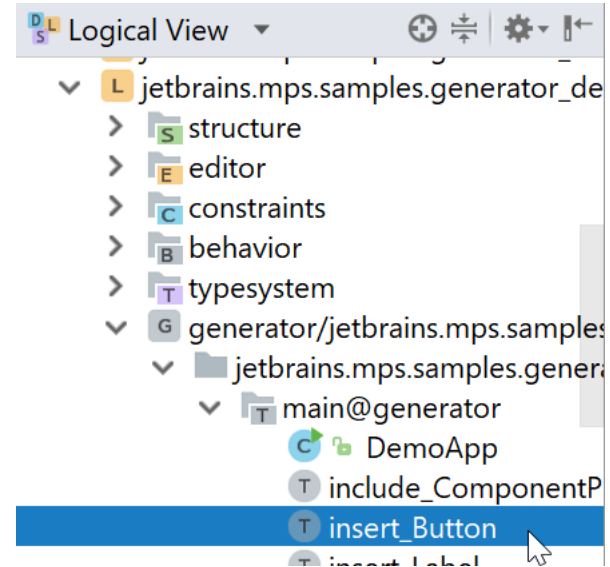
Input model

```
document Button
< button text = " Hello " enabled = " false " >
...
</ button >
```

Templates



- Inline template (directly inside a generator rule) → for producing a non-root node:



Input model

```
document Button
< button text = " Hello " enabled = " false " >
...
</ button >
```

Generator rule

```
[concept    Button
inheritors false
condition <always>]
```

Inline template

```
--> <T < button text = " $node.text :string " > T>
...
</ button >
```

Templates

- The actual template code is 'wrapped' in a **template fragment**. Any code outside template fragment is not used in transformation and serves as a context (for example you can have a Java class, but export only one of its method as a template).

```
template insert_Button
input Element

parameters
<< ... >>
```

```
content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
}
```

Fragment

<TF method

```
public static
    Component $genContext.unique name from (templateValue) in context (<no node>) :string() {
        JButton component = new JButton();
        if( node.attribute.findFirst({~it => it.name.equals("text") }).isNull )<T
            component.setText("$node.attribute.findFirst({~it => it.name.equals("text") }).value");
        T>
        $INCLUDE$ { node } → include_ComponentProperties
        return component;
    }
```

TF>

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - **Template elements: macros**
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- Mapping labels
- Generation context
- The generator algorithm
- Mapping scripts

Macros - property

- The code in templates can be parameterized through **macros**. The generator language defines three kinds of macros:
 - **property macro** - computes a property value;
 - **reference macro** - computes the target (node) of a reference;
 - **node macro** - used to control template filling at generation time. There are several versions of node macro: IF, LOOP, INCLUDE, CALL, SWITCH, COPY-SRC, COPY-SRCL, MAP-SRC, MAP-SRCL, and WEAVE.
- Property macro:

Input model

document Button

```
< button text = " Hello " enabled = " false " >  
...  
</ button >
```

```
concept Document extends BaseConcept  
implements INamedConcept  
  
instance can be root: true  
alias: <no alias>  
short description: <no short description>  
  
properties:  
<< ... >>  
  
children:  
rootElement : Element[1]  
  
references:  
<< ... >>
```

interface concept INamedConcept extends <none>

properties:
name : string

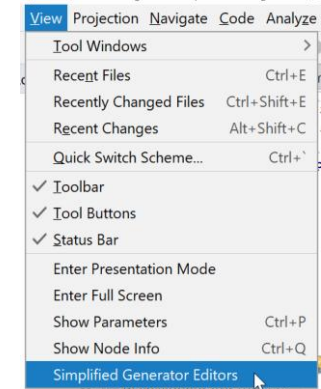
children:
<< ... >>

references:
<< ... >>

```
root template  
input Document  
  
public class $node.name :string {  
    public static void main(string[] args) {  
        JFrame frame = new JFrame("Demo");  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container container = frame.getContentPane();  
        container.setLayout(new FlowLayout());  
        container.add($SWITCH$ { node.rootElement } → switch_JComponentByElementName);  
        frame.pack();  
        frame.setLocationRelativeTo(null);  
        frame.setVisible(true);  
    }  
}
```

Macros – two views

- MPS-specific default notation of macros quite different from many other (especially text-based) notations, using the inspector a lot.
For people that don't like this, there is now a (proof-of-concept) simplified view that makes it more like many others:



Normal view

- We will be using mostly the normal view.

```
root template
input Document

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

Inspector

rails.mps.lang.generator.structure.PropertyMacro

property value

```
comment : <none>
value : (templateValue, genContext, node, operationContext)->string {
    node.name;
}
```

Simplified view

```
root template
input Document

public class $node.name :string {
    public static void main(string[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        container.add($SWITCH$ { node.rootElement }-> switch_JComponentByElementName);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

Macros - reference

- Reference macro (very similar to property macro) → computes an actual reference in the output model (used together with mapping labels):

Input model

document Button

```
< button text = " Hello " enabled = " false " >
...
</ button >
```

label **method** : Element -> StaticMethodDeclaration

template **insert_Button**
input Element

parameters
<< ... >>

content node:

```
public class _class_ {
  public _class_() {
    <no statements>
  }
  <TF method
  public static Component ${createComponent}() {
    JButton component = new JButton();
    $IF$[component.setText("${text}");]
    $INCLUDE$ include_ComponentProperties[]
    return component;
  }
  </TF>
```

template **insert_Label**
input Element

parameters
<< ... >>

content node:

```
public class _class_ {
  public _class_() {
    <no statements>
  }
  <TF method
  public static Component ${createComponent}() {
    JLabel component = new JLabel();
    $IF$[component.setText("${text}");]
    $INCLUDE$ include_ComponentProperties[]
    return component;
  }
  </TF>
```

Root template

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

Inspector of reference macro

reference target

comment : <none>

```
referent : (outputNode, genContext, operationContext, node) -> join(node<MethodDeclaration> | string | node-ptr<MethodDeclaration>)
{
  genContext.get output method for (node);
}
```


Macros - node

- IF macro → The wrapped template code is applied only if the condition is true. Otherwise the template code is ignored and an 'alternative consequence' (if any) is applied:



The screenshot displays the 'insert_Label' macro template in an IDE. The template is defined as follows:

```
template insert_Label
input Element

parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF method public static Component ${createComponent}() { TF>
        JLabel component = new JLabel();
        $IF$[component.setText("${text}");]
        $INCLUDE$ include_ComponentProperties[]
        return component;
    }
```

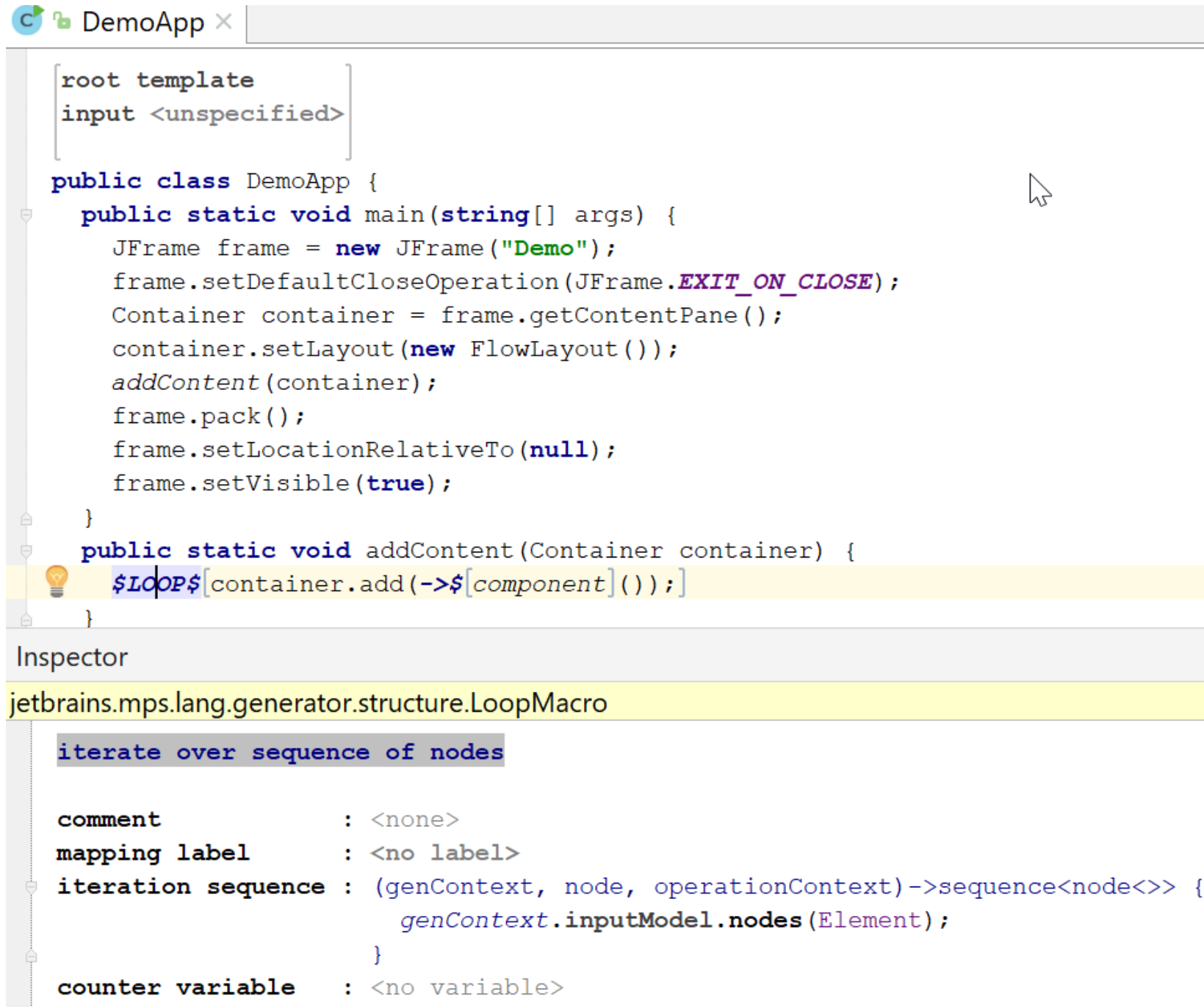
The 'Inspector' panel shows the configuration for the 'jetbrains.mps.lang.generator.structure.IfMacro' macro:

```
conditional branch

comment      : <none>
mapping label : <no label>
condition    : (genContext, node, operationContext)->boolean {
    node.attribute.findFirst({~it => it.name.equals("text"); }).isNotNull;
}
alternative  : <none>
```

Macros - node

- LOOP macro →
Computes new input
nodes and applies the
wrapped template to
each of them:



```
[root template
input <unspecified>]

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

Inspector

jetbrains.mps.lang.generator.structure.LoopMacro

iterate over sequence of nodes

comment : <none>

mapping label : <no label>

iteration sequence : (genContext, node, operationContext)->sequence<node<>> {
 genContext.inputModel.nodes(Element);
}

counter variable : <no variable>

Macros - node

- INCLUDE macro → The wrapped template code is ignored (it only serves as an anchor for the INCLUDE-macro), a reusable external template will be used instead:
- Null input makes INCLUDE effectively a no-op.

```
parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF method>
    public static Component ${createComponent}() {
        JLabel component = new JLabel();
        $IF${component.setText("${text}");}
        $INCLUDE$ include_ComponentProperties[]
        return component;
    }
}

Inspector
jetbrains.mps.lang.generator.structure.IncludeMacro

include outcome of a template

comment      : <none>
mapping label : <no label>
use input    : <current input node>

include template : include_ComponentProperties

template include_ComponentProperties
input      Element

parameters
<< ... >>

content node:
JComponent component = null;
<TF {
    $MAP_SRC${component.setEnabled(${false});}
    $MAP_SRC${
        component.setOpaque(true);
        component.setBackground(Color.->${black});
    }
}
```

```
template insert_Button
input      Element

parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF method>
    public static Component ${createComponent}() {
        JButton component = new JButton();
        $IF${component.setText("${text}");}
        $INCLUDE$ include_ComponentProperties[]
        return component;
    }
}

Inspector
jetbrains.mps.lang.generator.structure.IncludeMacro

include outcome of a template

comment      : <none>
mapping label : <no label>
use input    : <current input node>

include template : include_ComponentProperties
```

Macros - node

- CALL macro → Invokes template and replaces wrapped template code with the result of template invocation. Supports templates with parameters.
- Null input node is tolerated, and the template is ignored altogether in this case, i.e. CALL yields empty collection of nodes as a result when input/mapped node is null.

```
template insert_Button
input Element

parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF method>
    public static Component $[createComponent]() { TF>
        JButton component = new JButton();
        $IF$[component.setText("$[text]");]
        $CALL$ include_ComponentProperties[]
        return component;
    }
}
```

Inspector

jetbrains.mps.lang.generator.structure.TemplateCallMacro

call template and insert its outcome

comment : <none>
mapping label : <no label>
mapped node : <current input node>

template : include_ComponentProperties ("myStringValue")

```
template
input
Element

parameters
myParam : string

content node:
JComponent component = null;
<TF> {
    $MAP_SRC$[component.setEnabled($[false]);]
    $MAP_SRC$[
        component.setOpaque(true);
        component.setBackground(Color.->$[black]);
    ]
}
```

Macros - node

- SWITCH macro (used with a template switch) → Provides a way to many alternative transformations in the given place in the template code.
- The wrapped template code is applied, if none of switch cases is applicable and no default consequence is specified in template switch.
- For null input node, SWITCH may react with a message (specified along with its rules), anchor template node is ignored, and SWITCH macro yields no results.

```
switch_JComponentByElementName ×
template switch switch_JComponentByElementName extends <none>
parameters
<... >>

null-input message: <none>

cases:

[concept Element
inheritors false
condition (genContext, node, operationContext)->boolean {
node.name.equals("button");
}] --> <T> new JButton() <T>

[concept Element
inheritors false
condition (genContext, node, operationContext)->boolean {
node.name.equals("label");
}] --> <T> new JLabel() <T>

default: DISMISS TOP RULE error : 'button' or 'label' element name is expected
```

```
map_Document ×
[ root template
input Document

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

Inspector

jetbrains.mps.lang.generator.structure.TemplateSwitchMacro

switch templates by input node

comment	: <none>
mapping label	: <no label>
use input	: (genContext, node, operationContext)->node<> { node.rootElement; }

Macros - node

- COPY_SRC macro
→ Copies an input node to the output model. The wrapped template code is ignored.

```
template    reduce_Element
input      Element

parameters
< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    public static void _method_() {
        <TF [ ->$_method_() ] TF>;
    }
}
```

```
insert_Panel x
template    insert_Panel
input      Element

parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF factory_method
    public static Component $[createComponent]() {
        JPanel component = new JPanel();
        $INCLUDE$ include_ComponentProperties[]
        // add children
        $LOOP$[component.add($COPY_SRC$[null]);]
        return component;
    } TF>
```

Inspector

jetbrains.mps.lang.generator.structure.CopySrcNodeMacro

copy/reduce node

```
comment      : <none>
mapping label : <no label>
mapped node   : <current source node>
```

Macros - node

- COPY_SRCL macro → same a COPY_SRC, but can be applied to an list of nodes instead of only one node.
- \$COPY_SRCL[...] can be seen as syntactic sugar for \$LOOP[\$COPY_SRC[...]]

The image shows a screenshot of an IDE with two macro definitions and their corresponding inspectors.

Top Macro (insert_Panel):

```
parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF> factory_method
}
```

Inspector for insert_Panel:

```
jetbrains.mps.lang.generator.structure.CopySrcListMacro

copy/reduce list of nodes

comment      : <none>
mapping label : <no label>
mapped nodes  : (genContext, node, operationContext)->sequence<node<>> {
    node.content;
}
```

Bottom Macro (insert_Panel):

```
parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF> factory_method
}
```

Inspector for insert_Panel:

```
jetbrains.mps.lang.generator.structure.CopySrcListMacro

copy/reduce list of nodes

comment      : <none>
mapping label : <no label>
mapped nodes  : (genContext, node, operationContext)->sequence<node<>> {
    node.content;
}
```

Bottom Macro (insert_Panel):

```
parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    <TF> factory_method
}
```

Inspector for insert_Panel:

```
jetbrains.mps.lang.generator.structure.CopySrcListMacro

copy/reduce list of nodes

comment      : <none>
mapping label : <no label>
mapped nodes  : (genContext, node, operationContext)->sequence<node<>> {
    node.content;
}
```

Macros - node

- MAP_SRC macro → Multifunctional macro, can be used for:
 - marking a template code with a mapping label; (note: \$LABEL macro can be used for this instead)
 - replacing the current input node with a new one;
 - perform a non-template based transformation;
 - accessing the output node for some reason.
- \$MAP_SRCL[...] is syntactic sugar for \$LOOP[\$MAP_SRC[...]]

include_ComponentProperties x

```
template include_ComponentProperties
input Element

parameters
<< ... >>

content node:
JComponent component = null;
<TF> {
  $MAP_SRC$ component.setEnabled($[false]);
  $MAP_SRC$ {
    component.setOpaque(true);
    component.setBackground(Color.->$[black]);
  }
}
```

Inspector

jetbrains.mps.lang.generator.structure.MapSrcNodeMacro

map node macro

```
comment      : <none>
mapping label : <no label>
mapped node  : (genContext, node, operationContext)->node<> {
                node.attribute.findFirst({~it => it.name.equals("enabled"); });
            }
```


Macros - node

- WEAVE macro → Allows to insert additional child nodes into the output model. The node wrapped in the WEAVE macro (or provided by the use input function) will have the supplied template applied to it and the generated nodes will be inserted.
- Invokes a specific weaving rule (generator rules treated at a later point in time)

I couldn't find any real usages of this macro, it seems people prefer use of weaving rules instead.

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - **Mapping configurations**
 - Types of generator rules
- Mapping labels
- Generation context
- The generator algorithm
- Mapping scripts

Mapping configurations



- Applicability of individual templates is defined by **generator rules**, which are grouped into **mapping configurations**
- A **mapping configuration** can form a single generation step, contains **generator rules**, defines **mapping labels** and may include **pre- and postprocessing scripts**
- Almost each generator rule consists of a **premise** and a **consequence**
- Nearly all rules contain a reference to the concept of the **input node** (or just input concept) in its premises. All rule premises also contain an optional **condition function**.
- Rule consequence commonly contains a reference to an **external template** (declared as a root node in the same or different model) or to an **in-line template** (conditional root rule and root mapping rule can only have reference to an external template).
- There are also several other versions of consequences.

mapping labels:

```
<< ... >>
```

parameters:

```
<< ... >>
```

is applicable:

```
<always>
```

conditional root rules:

```
<< ... >>
```

root mapping rules:

```
[concept      Document] --> map_Document
[inheritors   false]
[condition    <always>]
[keep input root default]
```

weaving rules:

```
<< ... >>
```

reduction rules:

```
<< ... >>
```

pattern rules:

```
<< ... >>
```

reduce references:

```
<< ... >>
```

abandon roots:

```
<< ... >>
```

drop attributes:

```
<< ... >>
```

pre-processing scripts:

```
<< ... >>
```

post-processing scripts:

```
<< ... >>
```

Project /Volumes/Proje...

View as: Logical View

- 1: Project
 - smodel
 - structure
 - editor
 - actions
 - constraints
 - behavior
 - typesystem
 - scripts
 - intentions
 - plugin
 - dataFlow
 - test
 - generator/BL
 - jetbrains.mps.lang.smodel.generator
 - main@generator
 - enum
 - operation
 - mc_assignmentStatement
 - mc_attributes
 - mc_concept_operations
 - mc_concept_property_opera
 - mc_implicitSelect
 - mc_link_operations
 - mc_linklist_operations
 - mc_main
 - mc_model_operations
 - mc_node_operations
 - mc_property_operations
 - mc_searchScope_operations
 - mc_type_internal
 - reduce_ConceptMethodCall_I
 - reduce_ConceptMethodCall_V
 - reduce_Concept_NewInstanc
 - reduce_CopyOperation
 - reduce_EnumMemberRefere

Mapping configuration nodes

Template nodes

mc_model_operations

mapping configuration mc_model_operations
top-priority group false

Mapping configuration editor

mapping labels:
<< ... >>

conditional root rules:
<< ... >>

root mapping rules:
<< ... >>

weaving rules:
<< ... >>

reduction rules:

Rule premise Rule consequence Input concept In-line template

[concept Model_GetLongNameOperation] --> <T> SModelOperations.getModelName(\$COPY_SRC\$[null]) <T>

[inheritors false]

[condition <always>]

[concept Model_CreateNewNodeOperation] --> reduce_Model_CreateNewNode

[inheritors false]

[condition <always>]

[concept Model_CreateNewRootNodeOperation] --> reduce_Model_CreateNewRootNode

[inheritors false]

[condition <always>]

[concept Model_AddRootOperation] --> <T> SModelOperations.addRootNode(\$COPY_SRC\$[null], \$COPY_SRC\$[null])

[inheritors false]

[condition <always>]

Condition function

[concept Model_RootsOperation] --> reduce_Model_RootsOperation_hasConcept

[inheritors false]

[condition (node, genContext, operationContext)->boolean {
 return node.concept != null;
}]

[concept Model_RootsOperation] --> reduce_Model_RootsOperation_noConcept

[inheritors false]

[condition (node, genContext, operationContext)->boolean {
 return node.concept == null;
}]

[concept Model_RootsIncludingImportedOperation]

[inheritors false]

[condition (node, genContext, operationContext)->boolean {
 return node.concept != null;
}]

External template

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - **Types of generator rules**
- Mapping labels
- Generation context
- The generator algorithm
- Mapping scripts

Rules – conditional root

- Generates a root node in the output model:
- Applied only one time (max) during a single generation step.

conditional root rules:

```
condition (genContext, operationContext)->boolean {  
    true;  
}  
--> : DemoApp
```

conditional root rules:

```
condition <always> --> : DemoApp
```

The screenshot shows the IntelliJ IDEA IDE interface. On the left, the 'Logical View' pane displays a project structure for 'jetbrains.mps.samples.generator_demo'. The 'main@generator' package is expanded, showing a 'DemoApp' class. A blue arrow points from the 'DemoApp' class in the Logical View to the 'DemoApp' code editor on the right. The code editor shows the following code:

```
[root template  
input <unspecified>]  
  
public class DemoApp {  
    public static void main(string[] args) {  
        JFrame frame = new JFrame("Demo");  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container container = frame.getContentPane();  
        container.setLayout(new FlowLayout());  
        $LOOP$[container.add($SWITCH$ switch_JComponentByElementName[null]);]  
        frame.pack();  
        frame.setLocationRelativeTo(null);  
        frame.setVisible(true);  
    }  
}
```

Rules – root mapping

- Generates a root node in the output model:

root mapping rules:

concept	Document	--> map_Document
inheritors	false	
condition	<always>	
keep input root	default	

The screenshot shows the MPS IDE interface. On the left, the 'Logical View' pane displays a project structure for 'jetbrains.mps.samples.generator_demo'. The 'main@generator' node is expanded, showing a 'map_Document' node. A blue arrow points from the 'map_Document' node in the Logical View to the 'map_Document' tab in the editor. The editor displays the 'root template' for the 'map_Document' rule, which includes an 'input Document' and a 'public class' definition for 'map_Document'.

```
root template
input Document

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

Rules – weaving

- Allows to insert additional child nodes into the output model:
- The rule is applied on each input node of the specified concept.
- The parent node for insertion should be provided by the **context** function.

weaving rules:

```
[concept      Element
inheritors   false
condition    (genContext, node, operationContext)->boolean {
               node.name.equals("button");
             }
] -->

[weave_Button
context      : (genContext, operationContext, node)->node<> {
               genContext.get output main-class for (node.ancestor<concept = XMLDocument>);
             }
]
```

label **main-class** : XMLDocument -> ClassConcept

root mapping rules:

```
[concept      XMLDocument
inheritors    false
condition     <always>
keep input   root default
] --> main-class : DemoApp
```

ceptor

ains.mps.lang.generator.structure.Root_MappingRule

root mapping rule

mapping label **main-class**

description <document rule's intentions here>

Rules – weaving

- Allows to insert additional child nodes into the output model:
- The rule is applied on each input node of the specified concept.
- The parent node for insertion should be provided by the **context** function.

weaving rules:

```
concept      Element
inheritors   false
condition    (genContext, node, operationContext)->boolean {
    node.name.equals("button");
}
```

```
weave_Button
context      : (genContext, operationContext, node)->node<> {
    genContext.get output main-class for (node.ancestor<concept = XMLDocument>);
}
```

template input **weave_Button**
Element

parameters
<< ... >>

content node:
public class _class_ {
<TF> \$LABEL\$ createComponentMethods

```
root template
input <unspecified>

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

```
public static Component $[createComponent]() {
    JButton component = new JButton();
    $IF$[component.setText("$[text]");]
    return component;
} <TF>
```

Rules – reduction

- Transforms the input node while this node is being copied to the output model:

```
template reduce_Element
input Element

parameters
<< ... >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    public static void _method_() {
        <TF [ ->$_method_() ] TF>;
    }
}
```

Inspector

jetbrains.mps.lang.generator.structure.ReferenceMacro

reference target

comment : <none>

referent : (outputNode, genContext, operationContext, node)->{

genContext.get output factory_method for (node);

reduction rules:

concept	Element	-->	reduce_Element
inheritors	false		
condition	<always>		

document Button

```
< button text = " Hello " enabled = " false " >
...
</ button >
```

Inspector

jetbrains.mps.sampleXML.structure.Element

```
root template
input <unspecified>

public class DemoApp {
    public DemoApp() {...}
    public static void main(string[] args) {...}
    public static void addContent(Container container) {
        $LOOP$[container.add($COPY_SRC$[null]);]
    }
}
```

```
public static void addContent(Container container) {
    container.add(createComponent_a());
    container.add(createComponent_a_0());
    container.add(createComponent_a_1());
}
```

document Button

Rules – pattern

- Transforms the input node, which matches the pattern:
- Similar to reduction rules, but premise is pattern instead of concept.
- Patterns are specified in terms of the input language.

pattern rules:

```
pattern > << button text = " Hello " enabled = " false " >> < --> reduce_Element
...
</ button >
condition <always>
```

```
< button text = " Hello " enabled = " false " >
...
</ button >
```

```
template reduce_Element
input Element

parameters
<< . . >>

content node:
public class _class_ {
    public _class_() {
        <no statements>
    }
    public static void _method_() {
        <TF [->$_method_()] TF>;
    }
}
```

Inspector

jetbrains.mps.lang.generator.structure.ReferenceMacro

reference target

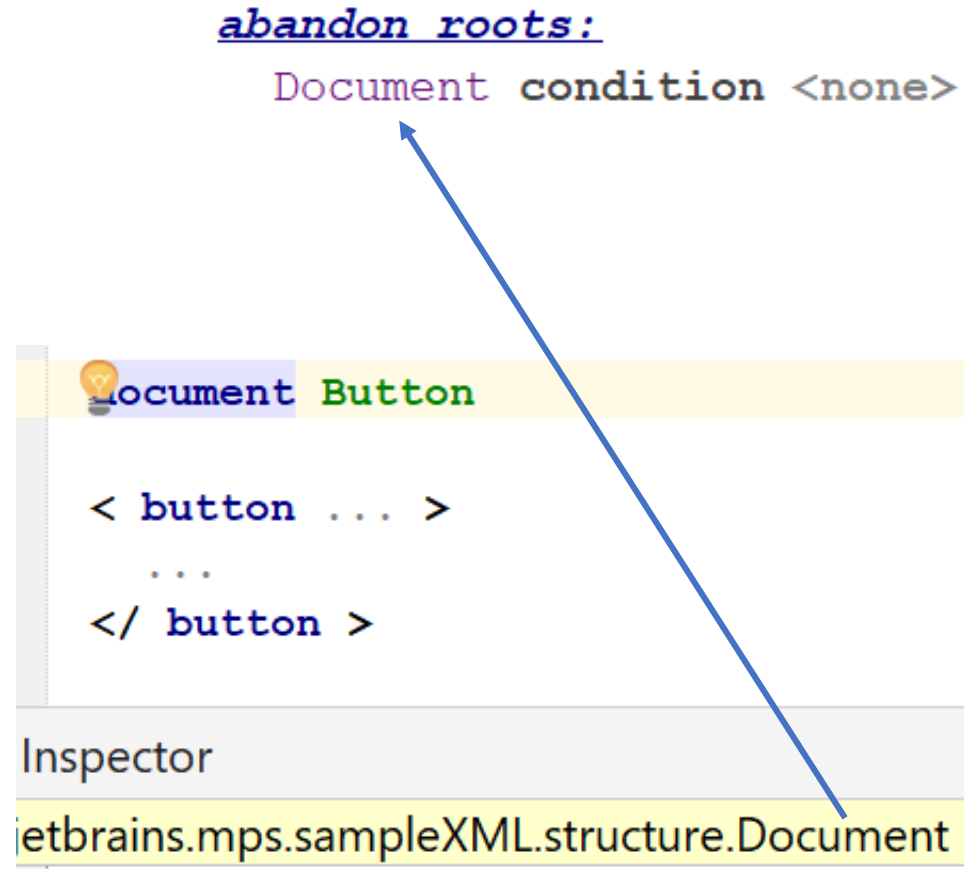
comment : <none>

referent : (outputNode, genContext, operationContext, node)->{

genContext.get output factory_method for (node);

Rules – abandon root

- Allows to drop an input root node which otherwise would be copied into the output model:



Rules – drop attribute

- For a transformed node, controls which attributes get copied from the input node:

```
concept Element extends ElementPart
    implements <none>
```

```
instance can be root: false
alias: <
short description: element
```

```
properties:
<< ... >>
```

```
children:
attribute : Attribute[0..n]
content   : ElementPart[0..n]
```

```
references:
<< ... >>
```

```
@attribute info
multiple: <inherited>
role: <inherited>
attributed concepts: Element
concept ElementAnnotation extends NodeAttribute
    implements <none>
```

```
instance can be root: false
alias: <no alias>
short description: <no short description>
```

```
properties:
<< ... >>
```

```
children:
<< ... >>
```

```
references:
```

```
💡 << ... >>
```

drop attributes:

```
ElementAnnotation condition <none>
```

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- **Mapping labels**
- Generation context
- The generator algorithm
- Mapping scripts

Mapping labels

- Mapping Labels are declared in a mapping configuration and references stored to this declaration are used to label generator rules, macros and template fragments. Such marks allow finding of an output node by a known input node. → N.B.: it's also possible to lookup things by smodel (if in scope), but mapping labels are most robust

root mapping rules:

```
concept      XMLDocument --> main-class : DemoApp
inheritors   false
condition    <always>
keep input  root default
```

```
root template
input <unspecified>
```

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

```
$LABEL$ addContentMethod
```

```
public static void addContent(Container container) {
  <no statements>
}
```

mapping labels:

```
label main-class      : XMLDocument -> ClassConcept
label addContentMethod : XMLDocument -> StaticMethodDeclaration
label createComponentMethods : Element -> StaticMethodDeclaration
```

```
template weave_Button
input      Element
```

```
parameters
<< ... >>
```

```
content node:
```

```
public class _class_ {
  <TF> $LABEL$ createComponentMethods
```

```
public static Component $[createComponent]() {
  JButton component = new JButton();
  $IF$[component.setText("$[text]");]
  return component;
}
</TF>
```

```
}
```

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- Mapping labels
- **Generation context**
- The generator algorithm
- Mapping scripts

Generation context

- Generation context (the *genContext* parameter in macro- and rule-functions) allows finding of nodes in the output model, generating unique names and provides other useful functionality.
- Generation context can be used not only in the generator models, but also in utility models (i.e. root nodes declared in models outside of the generator model, but used in it) - as a variable of type **gencontext**.
- Operations of *genContext* are invoked using the familiar dot-notation: *genContext.operation*. Categories of operations are:
 - Finding output node: usually on mapping labels and nodes
 - Generating unique name
 - Getting contextual info
 - Transferring user data: transient, step, or session object (see later, **generator algorithm**).

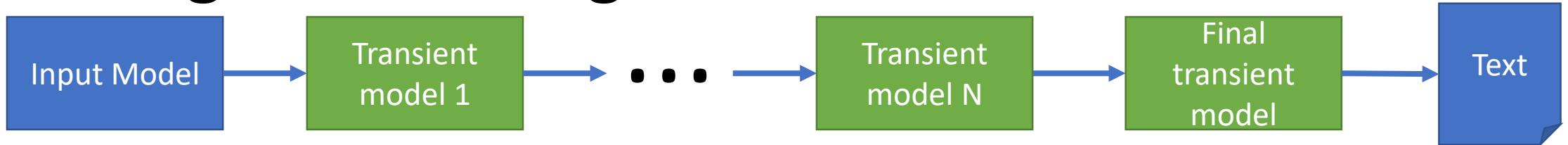
reference target

```
comment : <none>
referent : (outputNode, genContext, operationContext, node)->join(node<VariableDeclaration> | string | node-ptr<VariableDeclaration>)
{
    genContext.get output addContentMethod for (node.ancestor<concept = XMLDocument>).parameter.first;
}
```

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- Mapping labels
- Generation context
- **The generator algorithm**
- Mapping scripts

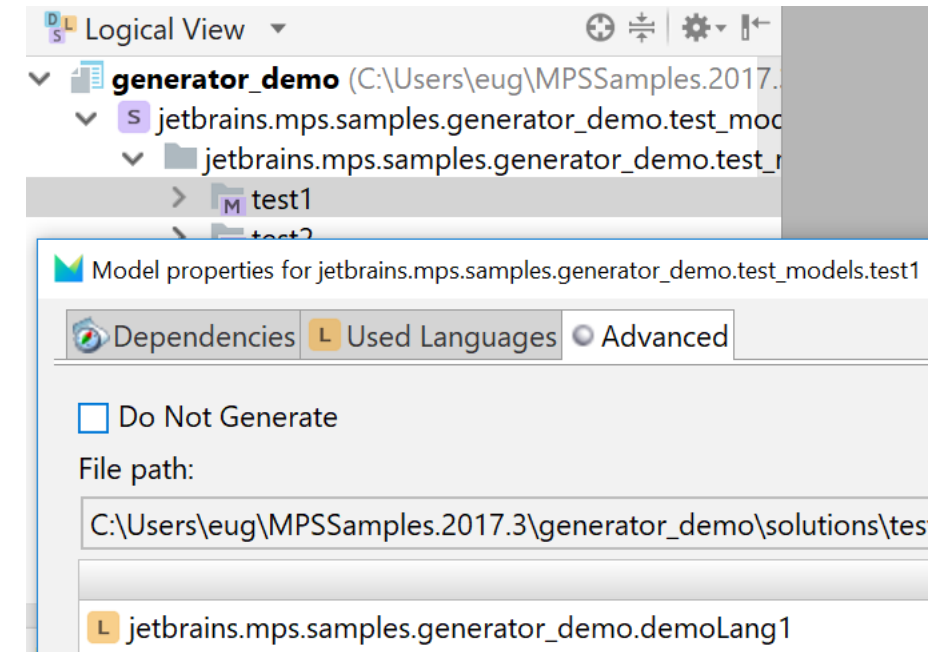
The generator algorithm



- Reduction is done in a bunch of transient models, until a fix-point has been reached
- The process of generation of target assets from an input model (generation session) includes 5 stages:
 - Defining all generators that must be involved
 - Defining the order of priorities of transformations
 - Step-by-step model transformation
 - Generating text and saving it to a file (for each root in output model)
 - Post-processing assets: compiling, etc.
- The first 3 stages live in model-to-model “land”, so we will explain them.

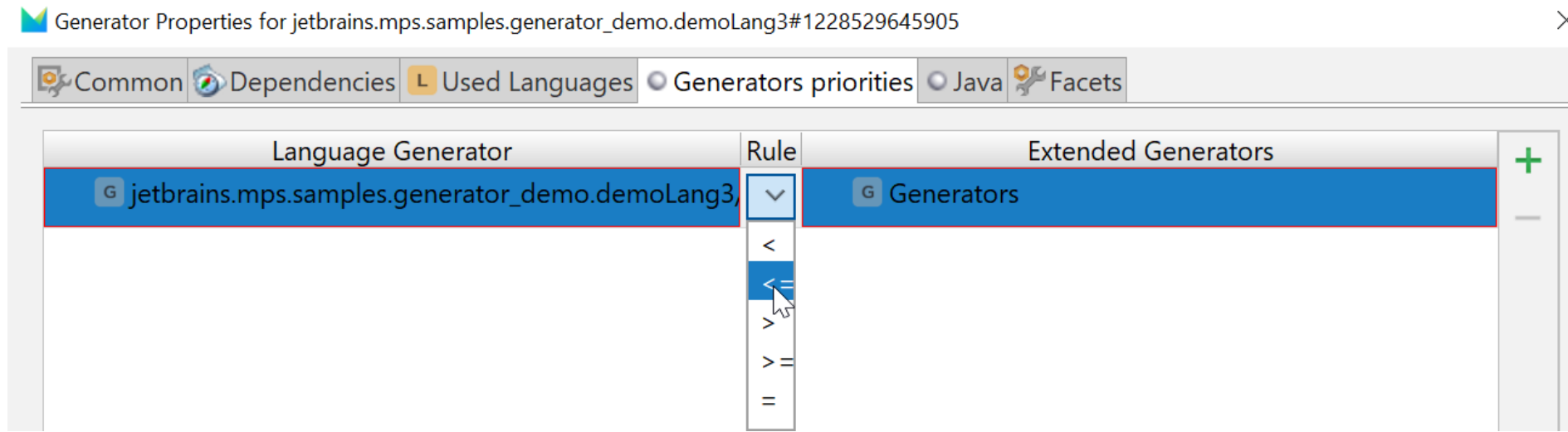
Defining generators involved

- MPS examines input model & determines which languages are used in it. MPS examines each node in the model and gathers languages that are actually used (so there is no reliance on user-specified language dependencies).
- From each 'used language' MPS obtains its generator module. Only 1 generator per module is supported. If any generator in this list depends on other generators (as specified in the 'depends on generators' property), those generators are added to the list as well.
- After MPS obtains the initial list of generators, it determines based on generator's templates what languages will be used in intermediate (transient) models. The languages detected this way are handled in the same manner as the languages used in the original input model. This procedure is repeated until no more 'used languages' can be detected.
- Explicit Engagement: In some rare cases, MPS is unable to detect a language whose generator must be involved in the model transformation. This may happen if that language is not used in the input model or in the template code of other (detected) languages. In this case, you can explicitly specify the generator engagement via the *Languages Engaged on Generation* section in the input model's properties dialog (*Advanced* tab).



Generator priorities

- It is often required that some mappings must be applied before (or not later than, or together with) some other mappings. Priority rules work on the **model** granularity level.
- The language developer specifies such a relationship between mappings by means of mapping constraints in the generator properties dialog.
- After MPS builds the list of involved generators, it divides all mappings into groups, according to the mapping priorities specified.
- All mappings for which no priority has been specified fall into the last (least-priority) group.
- MPS automatically inserts "not later than" rules for all **generator models** in the source and target languages (languages produced by templates).



Step-by-step model transformation

- Each group of mappings is applied in a separate **generation step**. The entire generation session consists of as many generation steps as there were mapping groups formed during the mapping partitioning. A generation step includes three phases:
 - Executing pre-mapping scripts
 - Template-based model transformation
 - Executing post-mapping scripts

Step-by-step model transformation

- The template-based model transformation phase consists of one or more micro-steps. A micro-step is a single-pass model transformation of an input model into a transient (output) model.
- Micro-steps are executed according to the following procedure:
 1. Apply conditional root rules (only once - on the 1-st micro-step)
 2. Apply root mapping rules
 3. Copy input roots for which no explicit root mapping is specified (this can be overridden by means of the 'keep input root' option in root mapping rules and by the 'abandon root' rules)
 4. Apply weaving rules
 5. Apply delayed mappings (from MAP_SRC macro)
 6. Revalidate references in the output model (all reference-macro are executed here)

Step-by-step model transformation

- There is no separate stage for the application of reduction and pattern rules. Instead, every time MPS copies an input node into the output model, it attempts to find an applicable reduction (or pattern) rule and performs the node copying when it is either copying a root node or executing a COPY_SRC-macro. Therefore, the reduction can occur at either stage of the model transformation.
- MPS uses the same rule set (mapping group) for all micro-steps within the generation step.
- After a micro-step is completed and some transformations have taken place during its execution, MPS starts the next micro-step and passes the output model of the previous micro-step as input to the next micro-step.
- The whole generation step is considered completed if no transformations have occurred during the execution of the last micro-step (fixpoint), that is, when there are no more rules in the current rule set that are applicable to nodes in the current input model.
- The next generation step (if any) will receive the output model of previous generation step as its input.

Model-to-model transformations

- Templates
 - Types of templates & template fragments
 - Template elements: macros
- Mapping configurations & generator rules
 - Mapping configurations
 - Types of generator rules
- Mapping labels
- Generation context
- The generator algorithm
- **Mapping scripts**

Mapping scripts

- A Mapping script is user code, which is executed either before a model transformation (pre-processing script) or after it (post-processing script).
- It should be referenced from the mapping configuration to be invoked as a part of it's generation step. Mapping scripts provide the ability to perform non-template based model transformations.
- Pre-processing scripts are commonly used for collecting certain information from input model that can be later used in the course of template-based transformation. The information collected by script is saved as a transient-, step- or session-object.

```
mapping script refine_text

script kind : post-process output model

(genContext, model, operationContext)->void {
    TextUtil.fixText(model);
}

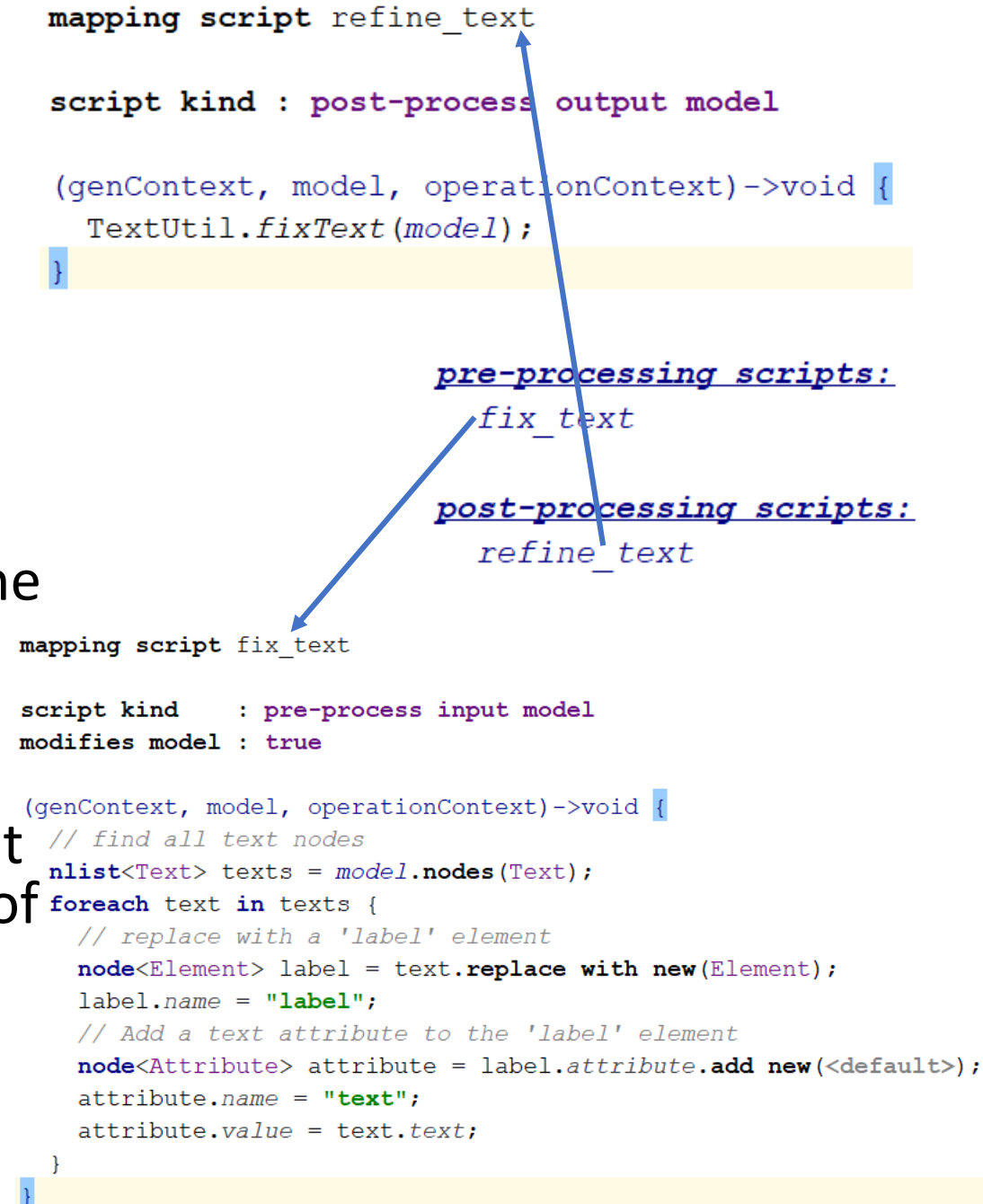
pre-processing scripts:
fix_text

post-processing scripts:
refine_text

mapping script fix_text

script kind      : pre-process input model
modifies model : true

(genContext, model, operationContext)->void {
    // find all text nodes
    nlist<Text> texts = model.nodes(Text);
    foreach text in texts {
        // replace with a 'label' element
        node<Element> label = text.replace with new(Element);
        label.name = "label";
        // Add a text attribute to the 'label' element
        node<Attribute> attribute = label.attribute.add new(<default>);
        attribute.name = "text";
        attribute.value = text.text;
    }
}
```



Index

- What are (MPS) generators?
- Model-to-model transformations
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- **Model-to-text transformations**
- Common generator patterns
- Basic dos and don'ts
- Further reading

Model-to-text transformations

- Textgen usually only used for very simple model-to-text scenarios where the cognitive gap between the model and the generated text is minimal (preferably cognitive gap is 0, i.e. model and text are 1:1), e.g.: translate a baseLanguage if-statement to a java if-statement or translate an attribute from an XML model to XML text

```
text gen component for concept IfStatement {  
  (node)->void {  
    append \n;  
    indent buffer;  
    append {if (} ${node.condition} {} {};  
    with indent {  
      append ${node.ifTrue};  
    }  
    append \n {} } $list{node.elsifClauses};  
    if (node.ifFalseStatement.isNotNull) {  
      append { else } ${node.ifFalseStatement};  
    }  
  }  
}
```

```
if (true) {  
  System.out.println("test");  
}
```

```
concept IfStatement extends Statement  
  implements IContainsStatementList  
  IDontSubstituteByDefault  
  IConditional  
  
instance can be root: false  
alias: if  
short description: <no short description>  
  
properties:  
  forceOneLine : boolean  
  forceMultiLine : boolean  
  
children:  
  condition : Expression[1]  
  ifFalseStatement : Statement[0..1]  
  ifTrue : StatementList[1]  
  elsifClauses : ElsifClause[0..n]  
  
references:  
<< ... >>
```

Model-to-text transformations & plaintextgen

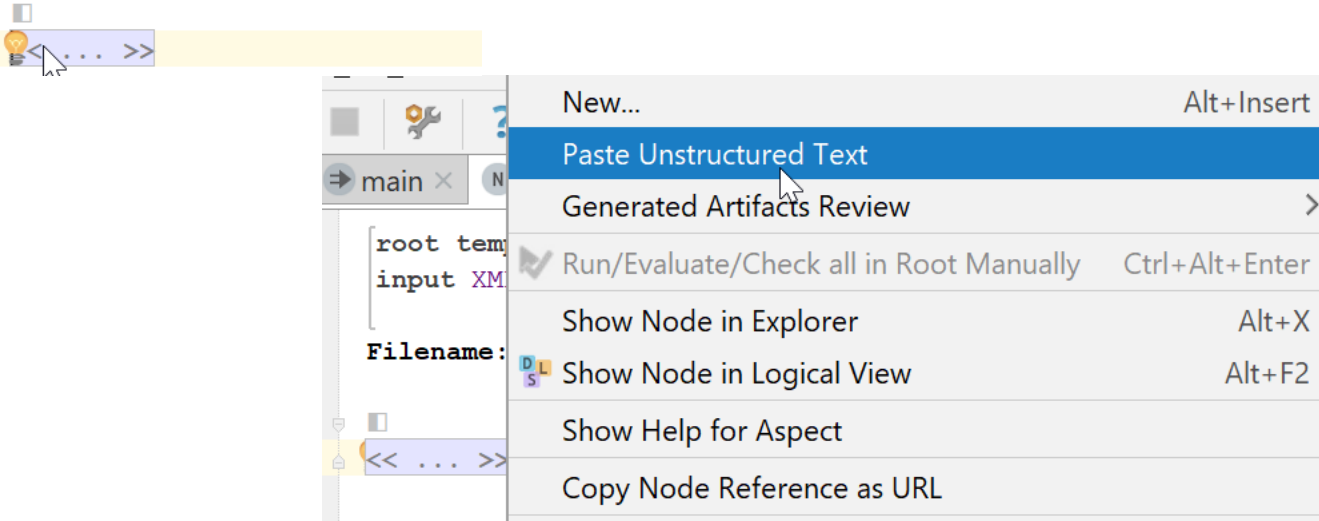
- Textgen usually only used for very simple model-to-text scenarios.
- For all other model-to-text scenarios, plaintextgen is recommended.
Examples of such scenarios:
 - For quickly getting text out of any model on any abstraction level
 - For generating to GPLs or horizontal DSLs (e.g. markup languages) that are not yet fully modeled in MPS
- Plaintextgen plugin encapsulates the textgen aspect and emulates textual-editor appearance and models the elements of plaintext in a certain flexible way that allows more maintainable and understandable text generators
- General advice: always use only plaintextgen and just skip textgen altogether

Typical plaintextgen scenario

- Create textual template
- Paste into analyzer
- Templatize

```
[root template  
input XMLDocument]
```

Filename: map_XMLDocument.xml

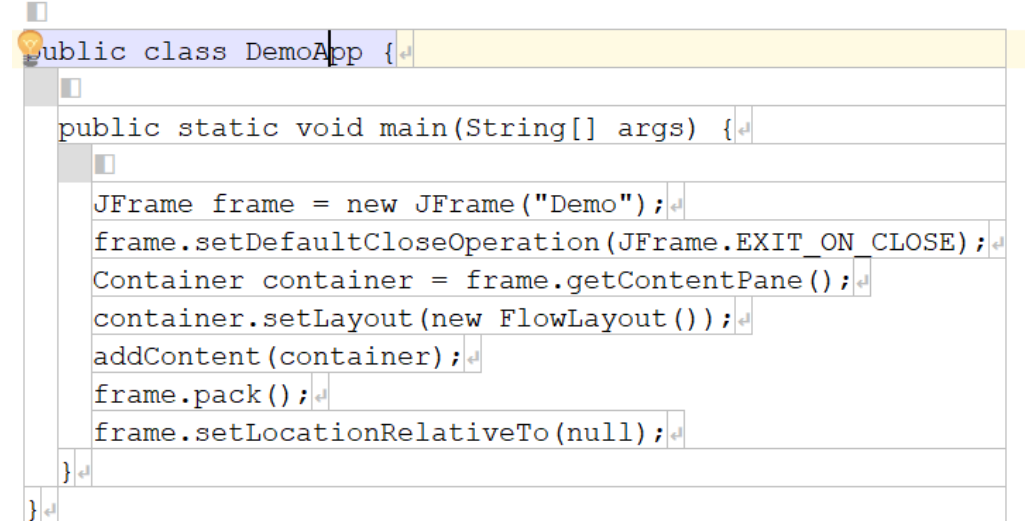


```
[concept      XMLDocument] --> map_XMLDocument  
[inheritors   false]  
[condition    <always>]  
[keep input root default]
```

```
public class DemoApp {  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("Demo");  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container container = frame.getContentPane();  
        container.setLayout(new FlowLayout());  
        addContent(container);  
        frame.pack();  
        frame.setLocationRelativeTo(null);  
    }  
}
```

```
[root template  
input XMLDocument]
```

Filename: map_XMLDocument.xml



Index

- What are (MPS) generators?
- Model-to-model transformations
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- Model-to-text transformations
- **Common generator patterns**
- Basic dos and don'ts
- Further reading

Common generator patterns

- **Detecting tests:** complex generators depending on some context may need to act differently in tests. `Com.mbeddr.mpsutil.blutil.genUtil` contains a **is-in-tests** construct detecting a test.
- **Preprocessing instead of reductions:** when the structures of source and target nodes are too different from each other, it may help to use pre/post processing scripts instead. In some cases, these are also easier to debug than unnecessarily convoluted
- **Introducing intermediate languages:** when the gap between source and target language is too big, it is helpful to introduce an intermediate language to reduce it. This also introduces an extra layer of decoupling, improving maintainability.

Common generator patterns

- **Error handling:** don't use exceptions (java) for error handling in generators. Instead, use `genContext.show error "myErrorMessage"`... This makes sure that generation doesn't stop immediately, improving debugging capabilities tremendously.
- **Reductions and extensibility:** if you need to provide many extension points for your generator, don't use a LOOP macro or do the transformation in place. Instead, use `COPY_SRC/COPY_SRCL` or `LOOP` in combination with a `SWITCH` macro that delegates to templates. Each of the places where you delegate to a template is a place where extensions can contribute/override with their own reduction rules.

Common generator patterns

- **SWITCH over IF:** When the condition of an IF macro is other than a boolean property, it is usually a smell that this IF should be replaced with a template switch. Using a SWITCH, makes the choice open for extension by the **extends** relation in each template switch.
- **(Currently) priority rules over predefined generation plans:** If extensibility is important for your generator, then use the priority rule mechanism to specify instead of the generation plan mechanism. When extension is not so important, generator plans can make things easier/faster to build.

Common generator patterns – multiple outputs from a single model

- These patterns apply for producing various outputs (e.g. XML, C++ code, java code, etc.) from a single model
- **Generator configuration:** the source model contains the actual contents to be generated, while there is/are (a) model(s) that are dedicated to the configuration of the generation. Example: mbeddr BuildConfiguration.
Such dedicated configuration models contain a reference to (part of) the contents-model. The generator of such a configuration model is responsible to copy all contents from the contents-model to the configuration model.
This also allows MPS to generate multiple outputs concurrently (as we have a single model per output we want to produce).

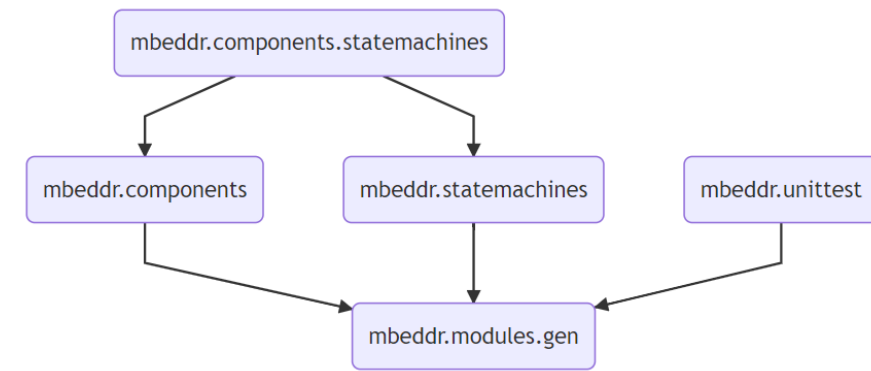
Common generator patterns – multiple outputs from a single model

- **Implementation of multi-output generators:** the generator of a dedicated configuration model needs to copy all relevant contents from the content-model. Instead of manually writing a lot of code that copies nodes, use *`copies = genContext.copy with trace`*. This makes MPS automatically take care of changing the references in the nodes that are copied. A big effort-saver.

Common generator patterns – complex multistage generators

- When writing large sets of generators and languages that build on top of each other it can be a challenge to understand what is going on during generation. Especially defining generator priorities to order them correctly can get messy quickly.
- If not taken care of it can happen that a lot of cross generator dependencies are introduced just for the sake of making sure generators are executed in the correct order.
- A pattern to counter this is to define logical barriers in your generator priorities. Priorities are then assigned relative to these barriers. The barriers represent levels of abstraction.
- If a generator requires a certain level of abstraction as input, its priorities are defined according to that. It is important a single generator is picked for each of these barriers to have a single point where these dependencies are relative to. This makes debugging much easier.
- Mbeddr Example: While mbeddr itself uses over 30 generators in total, their priorities are in most cases easy to understand. Most of the generators are isolated and most of them only define that they need to be run before the `modules.gen` generator.

Common generator patterns – complex multistage generators



- In this example we will look at 3 logical stages of mbeddr all of them on different layers of abstraction. They are explained from the bottom to the top.
- **mbeddr.modules.gen layer (lowest layer of abstraction):** This layer assumes that the input is mbeddr C99 representation, basically a simplified version of C99 without headers and some minor adaptations w.r.t. C99.
This generator transforms its input into real C99 code with .c and .h files. If a language extension (e.g. the mbeddr.unittest language) provides a higher abstraction than this, then it defines its generator priorities relative to this generator.
- **The middle layer:** This layer contains various abstractions on a higher level than C. They are all independent from each other, but at some point they need to generate down to mbeddr C. This needs to happen before the modules.gen generator is executed, because it assumes that the input is C. All of the languages define their priority relative the modules.gen generator. Due to this, debugging if the order is correct is easy. If transformations are not applied correctly, it is easy to check the generation plan for a model to see if all the generators reducing the abstraction to C have been executed before the modules.gen generator.
- **Higher Level Abstractions:** The top layer in our example is a language that integrates state-machines and components. Its priorities are only set relative to the two generators it extends: mbeddr.statemachines and mbeddr.components. Since these generators itself have priorities that require them to be executed before the modules.gen generator is run, no additional priorities are required.

Index

- What are (MPS) generators?
- Model-to-model transformations
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- Model-to-text transformations
- Common generator patterns
- **Basic dos and don'ts**
- Further reading

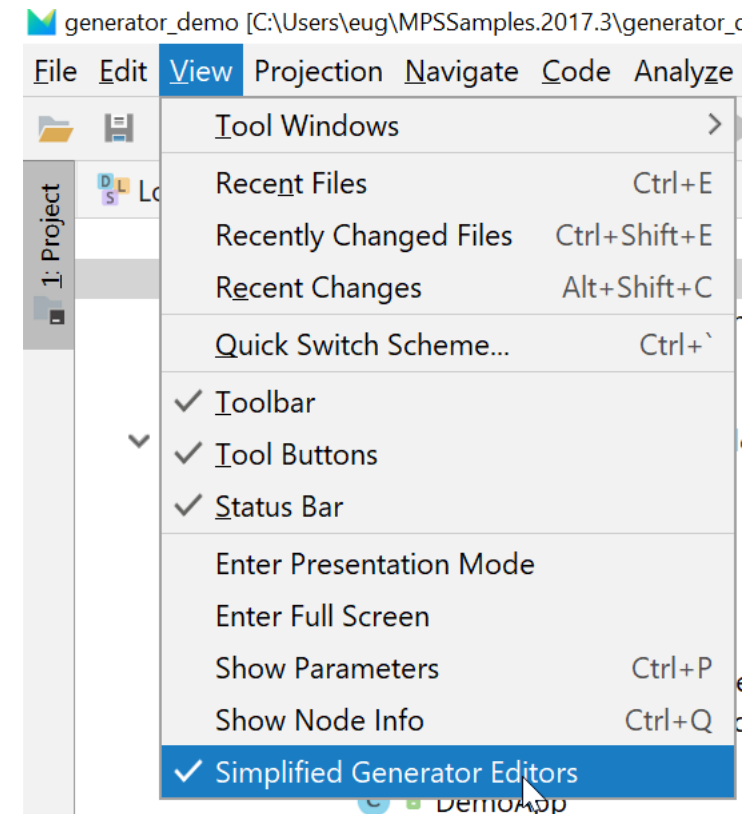
Dos and don'ts – good practices

- Tool

- If you don't like the default MPS way of displaying, use the simplified MPS generator editors plugin (<https://github.com/coolya/mps-generator-editors>) → currently only usable as a view, not editor
- To avoid any confusion, always follow this rule: after any changes made to the generator model, the model must be re-generated (*Shift+F9*). Even better is to use *Ctrl+F9*, which will re-generate all modified models in the generator module.
- Use “save transient models” for generator debugging
- Use generator trace tooling for debugging
- You can check the mapping partitioning for any (input) model by selecting *Show Generation Plan* action in the model's popup menu.

- Method

- A very good way to find/extract good examples is from the mbeddr codebase (use *Ctrl+N+N* to search for root nodes, explore the **mbeddr tutorial**, and look in the **modules pool** for example languages)



Index

- What are (MPS) generators?
- Model-to-model transformations
 - Templates
 - Mapping configurations & generator rules
 - Mapping labels
 - Mapping scripts
 - Generation context
 - The generator algorithm
 - Cross-model generation and generation plans
- Model-to-text transformations
- Common generator patterns
- Basic dos and don'ts
- **Further reading**

Further reading

- <https://confluence.jetbrains.com/display/MPSD20182/Generator>
- The MPS Language Workbench book (Fabien Campagne)
- Itemis guide on maintainable MPS generators: <https://coolya.github.io/maintainable-generators/>
- <https://confluence.jetbrains.com/display/MPSD20182/Generator+cookbook>
- <https://confluence.jetbrains.com/display/MPSD20182/Generator+User+Guide+Demo6#GeneratorUserGuideDemo6-savingtransientmodels>
- <https://confluence.jetbrains.com/display/MPSD20182/Generator+User+Guide+Demo6#GeneratorUserGuideDemo6-generationtracertool>
- <https://confluence.jetbrains.com/display/MPSD20182/Generator+Demos>
- <https://confluence.jetbrains.com/download/attachments/85756181/MPS%2B2017.1%2BCookbooks.pdf?version=1&modificationDate=1491299529000&api=v2>
- mbeddr tutorial, mbeddr languages (node infos?)