Editor Actions

MPS editor has quite sensible defaults in completion actions, node creation policy. But when you want to customize them, you have to work with the actions language.

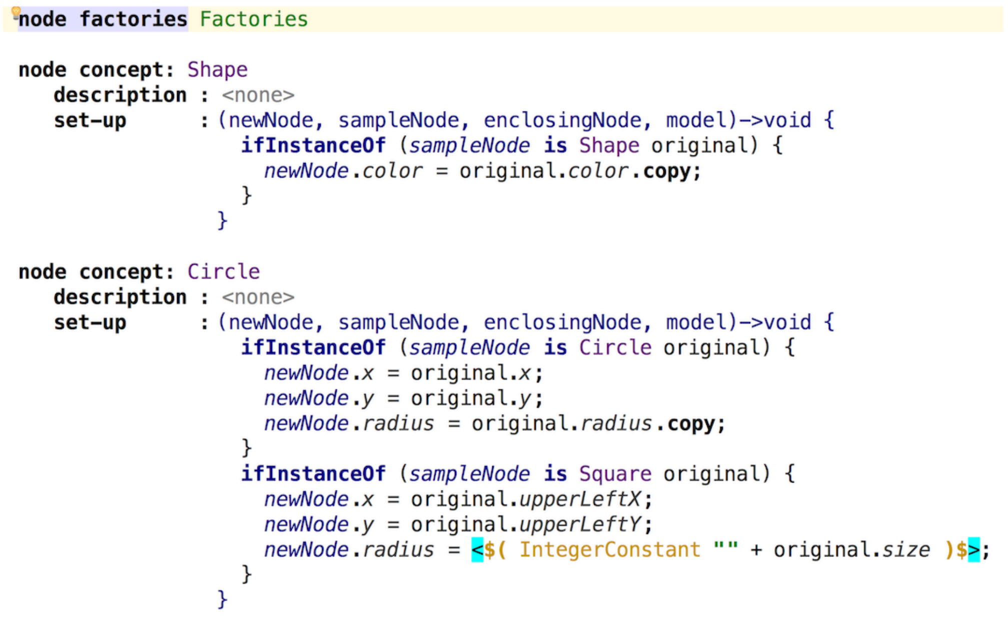
The Side-transformation actions as well as Node substitute actions have been deprecated in MPS 3.4 and replaced by the new Transformation Menu Language.

Node Factories

When a node is replaced with another one, it may be useful to parametrize the process of creation of the replacing node with values held by the node that is being replaced, or perhaps also to reflect the future position of the replacing node in the model. Node Factories give you exactly that. You write a set of handlers that get invoked whenever a new node needs to be created in a substitution action or through one of the new initialized node<>, set new initialized node<>, add new initialized node<>, replace with new initialized node<> and new initialized instance<> methods.

In brief, Node Factories allow you to customize instantiation of new nodes. In order to create node factory, you first have to create a new Node Factories root node. Inside of this root you can create node factories for concepts. Each node factory consists of node creation block which has the following parameters: newNode (the created node), sampleNode (the currently substituted node; can be null), enclosing node (a node which will be the parent of newNode in the model), and a model. The node factory handler is invoked before the new node gets inserted into the model.

You can leverage the concept inheritance hierarchy in Node Factories to reduce repetition.



```
node factories Factories

node concept: Shape
   description : <none>
   set-up      : (newNode, sampleNode, enclosingNode, model)->void {
                   ifInstanceOf (sampleNode is Shape original) {
                     newNode.color = original.color.copy;
                   }
                 }

node concept: Circle
   description : <none>
   set-up      : (newNode, sampleNode, enclosingNode, model)->void {
                   ifInstanceOf (sampleNode is Circle original) {
                     newNode.x = original.x;
                     newNode.y = original.y;
                     newNode.radius = original.radius.copy;
                   }
                   ifInstanceOf (sampleNode is Square original) {
                     newNode.x = original.upperLeftX;
                     newNode.y = original.upperLeftY;
                     newNode.radius = <$( IntegerConstant "" + original.size )$>;
                   }
                 }
```

To leverage node factories when creating nodes from code, use the "initialized" variants of "replace with ..." smodel la nguage constructs. See SModel language Modification operations for details.

Paste wrappers

These allow you to customize pasting of nodes into other contexts. For example, if you copy a LocalVariableDeclaration in Base Language and paste it into a ClassConcept to make it a field of the class, a simple transformation must be triggered that will create a new FieldDeclaration out of the LocalVariableDeclaration.

```
paste wrappers wrappers

paste wrapper Expression -> Statement
  (sourceNode)->node<Statement> {
    node<ExpressionStatement> result = new initialized node<ExpressionStatement>();
    result.expression.set(sourceNode);
    return result;
  }

paste wrapper ExpressionStatement -> Expression
  (sourceNode)->node<Expression> {
    return sourceNode.expression;
  }

paste wrapper LocalVariableDeclarationStatement -> LocalVariableDeclaration
  (sourceNode)->node<LocalVariableDeclaration> {
    return sourceNode.localVariableDeclaration;
  }

paste wrapper LocalVariableDeclaration -> Statement
  (sourceNode)->node<Statement> {
    node<LocalVariableDeclarationStatement> statement = new initialized node<LocalVariableDeclarationStatement>();
    statement.localVariableDeclaration.set(sourceNode);
    return statement;
  }

paste wrapper LocalVariableDeclaration -> ClassifierMember
  (sourceNode)->node<ClassifierMember> {
    node<FieldDeclaration> variable = new initialized node<FieldDeclaration>();
    variable.name.set(sourceNode.name);
    variable.type.set(sourceNode.type);
    variable.annotation.addAll(sourceNode.annotation);
    variable.isFinal.set(sourceNode.isFinal);
    return variable;
  }
```

Copy-paste handlers

These give you the possibility to customize the part of the models that is being copied to or pasted from the clipboard.

```
copy pre processor VariableReference
  (copy, original)->void {
    node<IVariableReference> qualifiedReference = copy.variableDeclaration.getQualifiedReference();
    if (qualifiedReference.isNotNull) {
      copy.replace with(qualifiedReference);
    }
  }
```

The copy parameter in a copy pre processor block gets contains an exact deep copy of the original parameter node. Unlike origi nal, copy is detached from the model and so has no parent node.

```
paste post processor StaticFieldReference
  (pastedNode)->void {
    if (Scope.parent(pastedNode) != null) {
      Scope surroundingScope = Scope.getScope(Scope.parent(pastedNode), pastedNode, ~~conceptNode~~/VariableDeclaration/);
      if (surroundingScope != null && surroundingScope.contains(pastedNode.getVariable())) {
        node<VariableReference> variableReference = pastedNode.replace with new initialized(VariableReference);
        variableReference.variableDeclaration = pastedNode.getVariable() : VariableDeclaration;
      }
    }
  }
```

The task for the paste post processor typically is to re-resolve references so that they point to declarations valid in the new context.

Previous Next